DS-GA 1007 | Lecture 3

Programming for Data Science

Jeremy Curuksu, PhD
NYU Center for Data Science
jeremy.cur@nyu.edu

September 27, 2022

DS-GA 1007 Curriculum

Programming for Data Science:

- Introduction to Programming in Python
- Best Practice Programming and Software Engineering
- **Program Efficiency**
- Interacting with Programs
- Array Manipulation for Scientific Computing
- Data Visualization
- Advanced Data Objects ($\times 4$)
- Environments for Collaborative Programming
- Industrial Applications

Program Efficiency

Program Efficiency

Last week:

- Modularization and Abstraction
- Functions and Objects in Python
- Testing and Debugging Programs

Today:

- **Program Run Time and Algorithmic Complexity**
- **Examples of Iterative and Recursive Algorithms**
- **Examples of Search and Sort Algorithms**

Program Run Time and Algorithmic Complexity

What is Program Efficiency?

How to compare the efficiency of two different programs?

What is Program Efficiency?

How to compare the efficiency of two different programs?

- How to measure the efficiency of an algorithm independent of machine or specific implemention?

What is Program Efficiency?

How to compare the efficiency of two different programs?

- How to measure the efficiency of an algorithm independent of machine or specific implemention?
- How to reason about an algorithm to predict the amount of time it will need to solve a problem of a particular size?

What is Program Efficiency?

How to compare the efficiency of two different programs?

- How to measure the efficiency of an algorithm independent of machine or specific implemention?
- How to reason about an algorithm to predict the amount of time it will need to solve a problem of a particular size?
- How to relate choices in algorithm design to time efficiency?

What is Program Efficiency?

How to compare the efficiency of two different programs?

- How to measure the efficiency of an algorithm independent of machine or specific implemention?
- How to reason about an algorithm to predict the amount of time it will need to solve a problem of a particular size?
- How to relate choices in algorithm design to time efficiency?

Example of algorithm to compute n^2:

```
def square(n):
    n2 = 0
    for i in range(n):
        for j in range(n):
            n2 += 1
    return n2
```

Isn't there a more efficient way?

Measures of a program's efficiency

Some options:

1. Measure the run time with a timer
2. Count the number of operations
3. Measure an order of growth as function of input size

Measure the run time of a program

Measure clock time on test data

- ✓ Run time varies between different algorithms
- × Run time varies between implementations
- × Run time varies between computers
- × Run time for large inputs not predictable based on small inputs

Example:

```
import time
t0 = time.time()
fact = 1
for i in range(1,1000):
   fact *= i
dt = time.time() - t0
print("Run time: {} seconds".format(dt))
```

Example of run time measurement

Measure run time with %timeit*

```
%timeit L = [n**2 for n in range(1000)]
```

Output:

```
148 µs ± 64 ns per loop (mean ± std dev of 7 runs, 10,000 loops each)
```

```
%timeit L = [square(n) for n in range(1000)]
```

Output:

```
7.34 s ± 60.5 ms per loop (mean ± std. dev. of 7 runs, 1 loop each)
```

*Details on Python magic commands:
jakevdp.github.io/PythonDataScienceHandbook/01.03-magic-commands.html

Counting Operations in a Program

Count number of operations as function of input size:

- ✓ #ops varies between different algorithms
- × #ops varies between implementations
- ✓ #ops does not vary between computers
- ✓ #ops for large inputs is predictable based on small inputs
- × No clear definition of which operations to count

Example:

```
def factorial(n):
    fact = 1                   1 ops
    for i in range(1,n+1):     1 ops
        fact *= i              2 ops
    return fact                => 1 + 3n ops
```

Generalization: Order of Growth

Count number of *key* operations in term of input size

- **#ops**: Measures an algorithm's run time in term of input size but does not measure scalability to arbitrarily large input size

- **Asymptotic Growth**: We need measure scalability to arbitrarily large input size, independent of specific implementation

- **Focus on Dominant Terms**: We need measure an invariant order of magnitude based on largest factors/bottlenecks

- **Lower/Upper Bound**: We can measure the average, best or worst case over all possible inputs of a given size

O() as Order of Growth of a Program

Upper Bound Asymptotic Growth relative to input size:

- ✓ $O()$ varies between algorithms, not implementations
- ✓ $O()$ measures rate of growth of run time as input size grows
- ✓ $O()$ is a tight upper bound on order of magnitude growth

Example:

```
def factorial(n):
    fact = 1                  1 ops
    for i in range(1,n+1):    1 ops
        fact *= i             2 ops
    return fact               => 1 + 3n ops
```

Ignore additive constants $=> 3n$ and multiplicative constants $=> n$

Worst case asymptotic complexity $= O(n)$

Analyzing Complexity of a Program

Order of Growth of a Program O():

- **Rule 1: Focus on dominant terms inside statements**: Drop additive and multiplicative constants

Examples:

- $1 + 3n$ has worst case complexity of $O(n)$
- $2 + 2n + n^2$ has worst case complexity of $O(n^2)$
- $10000 + 1000000n + n^2$ has worst case complexity of $O(n^2)$
- $n^5 + 5^n$ has worst case complexity of $O(5^n)$

Analyzing Complexity of a Program

Order of Growth of a Program O():

- **Rule 2: Use the law of addition for sequential statements**: Just *add up* orders of growth between consecutive operations $O(f(n) + g(n)) = O(f(n)) + O(g(n))$

Example:

```
for i in range(n):
   print(i)
for j in range(n*n):
   print(j)
```

- The first *for* loop has worst case complexity of $O(n)$
- The second *for* loop has worst case complexity of $O(n^2)$
- Worst case complexity $=> O(n) + O(n^2) = O(n^2)$

Analyzing Complexity of a Program

Order of Growth of a Program O():

- **Rule 3: Use the law of multiplication for nested statements**: Multiply orders of growth between nested loops because an inner loop is repeated for each outer loop iteration
  $O(f(n) \times g(n)) = O(f(n)) \times O(g(n))$

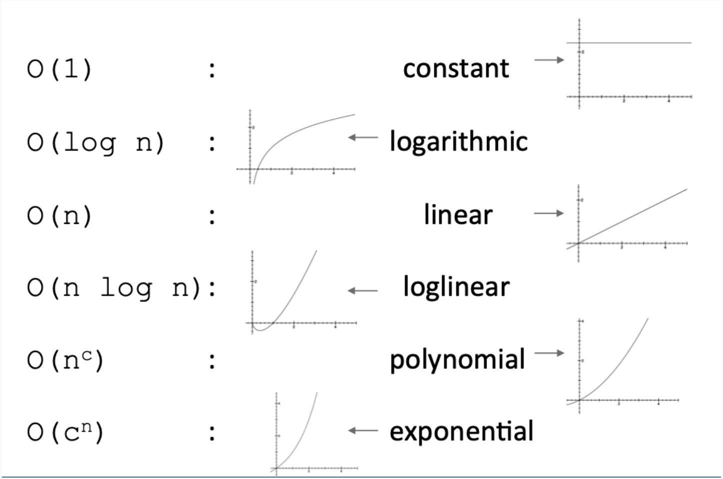
Example:

```
for i in range(n):
    for j in range(n):
        print(i,j)
```

- The first *for* loop has worst case complexity of $O(n)$
- The second *for* loop has worst case complexity of $O(n)$
- Worst case complexity $=> O(n) \times O(n) = O(n^2)$

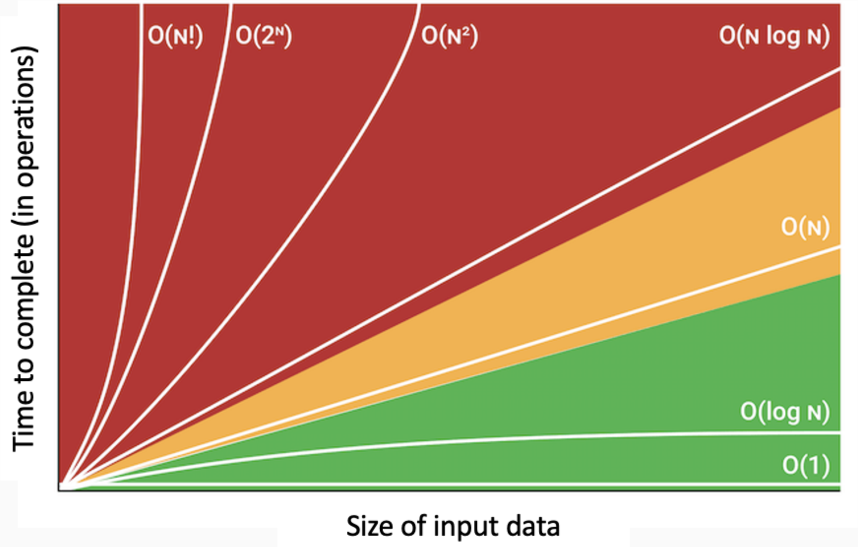
Map of Algorithmic Complexities

O() relates choices in algorithm design to time efficiency. The goal of the programmer is to write algorithms that go up in the map



O(1) :	constant
O(log n) :	logarithmic
O(n) :	linear
O(n log n):	loglinear
$O(n^c)$:	polynomial
$O(c^n)$:	exponential

Map of Algorithmic Complexities



Examples of Algorithm Complexity Analysis

Algorithm Complexity Analysis: x^2

Example of iterative nested loop to calculate x^2:

```
def square(x):
    x2 = 0
    for i in range(x):
        for j in range(x):
            x2 += 1
    return x2
```

Algorithm Complexity: $O(n^2)$

Algorithm Complexity Analysis: x²

Example of recursive function to calculate x^2

$$
\begin{aligned}
(x-1)^2 &= x^2 - 2x + 1 \\
<=> \quad x^2 &= (x-1)^2 + 2x - 1
\end{aligned}
$$

```
def square(x):
    if x == 0:
        return x
    else:
        return (square(x-1) + 2*x - 1)
```

Algorithm Complexity: $O(n)$

Algorithm Complexity Analysis: !x

Example of iterative loop to calculate !x:

```
def factorial(x):
    fact = 1
    for i in range(1,x+1):
        fact *= i
    return fact
```

Algorithm Complexity: $O(n)$

Algorithm Complexity Analysis: !x

Example of recursive function to calculate $!x$

```
def factorial(x):
    if x <= 1:
        return 1
    else:
        fact = x * factorial(x-1)
        return fact
```

Algorithm Complexity: $O(n)$

Search and Sort Algorithms

Linear Search of Element in List

Linear Search: Unsorted List

```
def search(L,e):
    found = False
    for i in range(len(L)):
        if L[i] == e:
            found = True
    return found
```

Algorithm Complexity: $O(n)$

(due to worst case scenarios)

Linear Search of Element in List

Linear Search: Sorted List

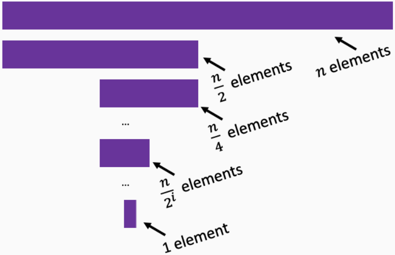
```
def search(L, e):
    for i in range(len(L)):
        if L[i] == e:
            return True
        if L[i] > e:
            return False
    return False
```

Algorithm Complexity: $O(n)$

(due to worst case scenarios)

Recursive Search of Element in List

Bisection Search



Search complete when
$\frac{n}{2^i} = 1 \ <=> \ i = log\, n$

Complexity: $O(log\, n)$

- ▶ Pick index that divides list in half, test if $L[m] == e$. If not, test if $L[m]$ is larger or smaller than e. Then, depending on the answer, search left or right half of L

- ▶ Bisection is a divide-and-conquer algorithm: Break original problem in smaller versions of the problem (smaller lists)

Recursive Search of Element in List

Bisection Search: Sorted List*

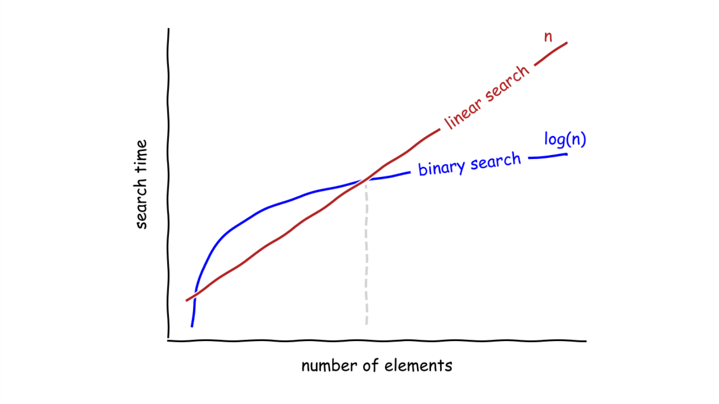
```
def bisection_search(L,e,low,high):
    if high == low:
        return L[low] == e
    mid = (low + high)//2
    if L[mid] == e:
        return True
    elif L[mid] > e:
        if low == mid:  # Nothing left to search
            return False
        else:
            return bisection_search(L,e,low,mid - 1)
    else:
        return bisection_search(L,e,mid + 1,high)
```

Algorithm Complexity: $O(\log n)$

**Sorting required*

Recursive Search of Element in List

Bisection Search is Faster for Large Lists



Sorting Algorithms

Is it Better to Sort before Searching?

- **Never True for Single Search**: Sorting a collection of elements requires to look at each one at least once

- **Amortize Cost**: Sort becomes beneficial for multiple searches: sort list once, then do many searches

- **Algorithmic Complexity of *k* searches**:
  $O(\text{sort}) + k \times O(log\, n) < k \times O(n)$?
  $=>$ When k is large, run time for sort may become irrelevant relative to run time for search

Sorting Algorithm: MonkeySort

MonkeySort: Randomly shuffle, repeat until sorted

```
def monkey_sort(L):
    while not is_sorted(L):
        random.shuffle(L)
```

Algorithm Complexity:

- Best case: $O(n)$
- Worst case: $O(\infty)$

Sorting Algorithm: SelectionSort

SelectionSort: For each index *i* in list, loop from *i* to end of list, find lowest element, swap with element at index *i*

```
def selection_sort(L):
    for i in range(len(L)):
        min_i = i
            for j in range(i+1, len(L)):
                if L[j] < L[min_i]:
                    min_i = j
        L[i], L[min_i] = L[min_i], L[i]
```

Algorithm Complexity: $O(n^2)$

Sorting Algorithm: BubbleSort

BubbleSort: Pass through pairs of elements, compare elements in each pair, swap so that smallest first, repeat

```python
def bubble_sort(L):
  swap = True
  while swap:
    swap = False
    for i in range(len(L)): # Pass through pairs
      if L[i] > L[i+1]: # Compare elements
        swap = True
        L[i], L[i+1] = L[i+1], L[i]
```

At each pass, largest unsorted element gets bubbled up to the end, so at most *n* passes in while loop $=>$ **Algorithm Complexity**: $O(n^2)$

Sorting Algorithm: MergeSort

MergeSort: Recursively split into half sublists, then sort sublists while merging them back together

```
def merge_sort(L):
    if len(L) < 2:
        return L[:]
    else:
        middle = len(L)//2
        left = merge_sort(L[:middle])
        right = merge_sort(L[middle:])
        return merge(left, right)
```

Fastest version (optional for this course)

MergeSort: Version less easy to read than on previous slide, but fastest (doesn’t copy sublists)

```
def merge_sort(L,l,h): # Start with l=1, h=len(L)
   if h == l:
      return L[l-1]
   else:
      mid = (l + h)//2
      left = merge_sort(L,l,mid)
      right = merge_sort(L,mid+1,h)
      return merge(left, right)
```

Algorithm Complexity: $O(\log n) \times$?

Merge Sorted Sublists

```python
def merge(left, right):
    result = []; i,j = 0,0
    while i < len(left) and j < len(right):
        if left[i] < right[j]:
            result.append(left[i])
            i += 1
        else:
            result.append(right[j])
            j += 1
    while (i < len(left)):
        result.append(left[i])
        i += 1
    while (j < len(right)):
        result.append(right[j])
        j += 1
    return result
```

Algorithm Complexity: $O(n)$

Sorting Algorithm: MergeSort

Divide and Conquer: $O(n\ log\ n)$

- **Divide using *merge_sort()***: $O(log\ n)$
  Divide lists into halves until each sublist contains only 1 element (which by definition are sorted) creates log(n) recursion levels $=>$ $O(log\ n)$ operations

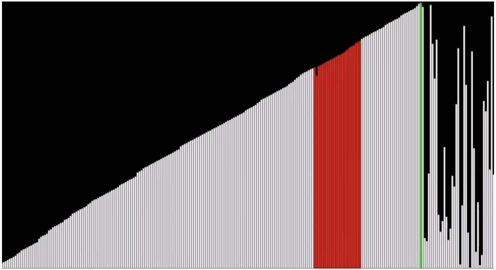
- **Conquer using *merge()***: $O(n)$
  At each recursion level, **sorting** pairs of sorted sublists is linear in number of elements (which is $2^i \times \frac{n}{2^i} = n$) because smallest elements are always the first elements. Copying (**merging**) each element takes exactly n operations $=>$ $O(n) + n$ operations

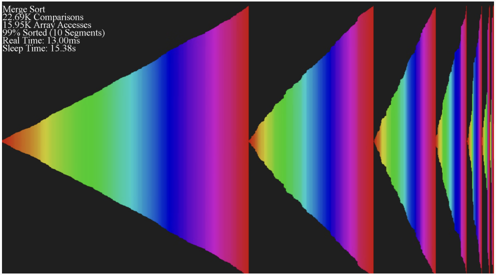
Sorting Algorithms: In a Nutshell

We looked at:

1. **Monkey sort:** $O(\infty)$
   Relies on randomness, potentially unbounded
2. **Selection sort:** $O(n^2)$
   Guarantees first i elements are sorted
3. **Bubble sort:** $O(n^2)$
   Guarantees last i elements are sorted & entire list gets *overall* more sorted at every step
4. **Merge sort:** $O(n \log n)$
   The fastest a sort can be...

Sorting Algorithms: In a Movie





```
Merge Sort
22.69K Comparisons
15.95K Array Accesses
99% Sorted (10 Segments)
Real Time: 13.00ms
Sleep Time: 15.38s
```

Thank you!