Northeastern University

CS6020: Collecting, Storing, and Retrieving Information

Basic Data Shaping

Basic Data Shaping

ASSESSING COMPLEXITY

Lesson Objectives

- After completing this lesson, you are able to:
  - appreciate the time and space complexity considerations for a function or program
  - assess the time and space complexity of an algorithm
  - understand the growth of space and time for an algorithm as a function of its input size
  - measure the performance of a function in R

What is Complexity?

- Complexity is a way to measure time and space required for an algorithm or program to execute.
- Algorithmic complexity is concerned about how fast or slow a particular algorithm performs.
- Complexities are used to compare algorithms on a conceptual level, *i.e.*, ignoring low level details.

Asymptotic Notation

- Complexity as a numerical function *T(n)*
  - time versus the input size *n*
- Time required to run depends on various factors such as processor speed, instruction set, disk speed, brand of compiler, *etc*.
- Consequently, we estimate time asymptotically, *i.e.*, the value it will eventually reach.
- We measure time *T(n)* as the number of elementary "steps”.

Example: Add 2 Integers

- Consider adding two binary integers digit by digit (or bit by bit)
  - adding a single bit is a "step" in the computation
- Adding two *n*-bit integers takes *n* steps.
- Consequently, the total computational time

  $T(n) = c * n$

  where *c* is the time taken for one addition step.

Execution Time vs Growth

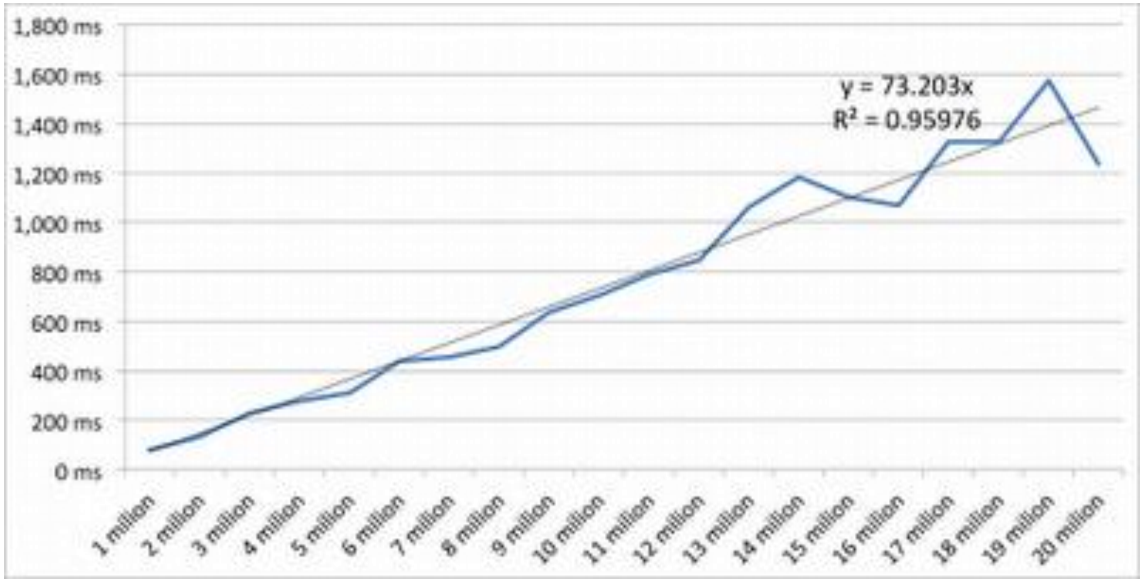
- The actual execution is difficult to estimate as it depends on a multitude of factors:
  - type of CPU
  - operating system
  - programming language
  - other processes running concurrently
  - data structures and data representation
- Programmers compare and classify algorithms through their asymptotic growth as a function of the size of the input.

Conducting Experiments

- Aside from estimating runtime and space complexity of a program through analysis of its algorithms, you can often understand its behavior through experiments.
- Measure how long the program takes to run as you increase the size of inputs, data objects, or data files.
- Use the `system.time()` function to time calls to functions.

Regression Analysis

- Once timing measurements are obtained, time can be plotted against input size and a regression curve can be fitted against the data.



Best, Worst, Average Cases

- There is often a significant difference between the best, worst, and average runtime of a program.
- For example, what if you needed to sort a data set by year?
  - if it’s already sorted it’ll be much faster than if it weren’t
- We generally look for average case behavior.

Example: Searching a List

- Suppose you need to search a column in a data frame or a vector for a specific element, *e.g.*, find the data for flight "JB721-010214".
- How many string comparisons would you need to perform if there were *n* elements?
  - on average: `n/2`
  - worst case: `n`
  - best case: `1`

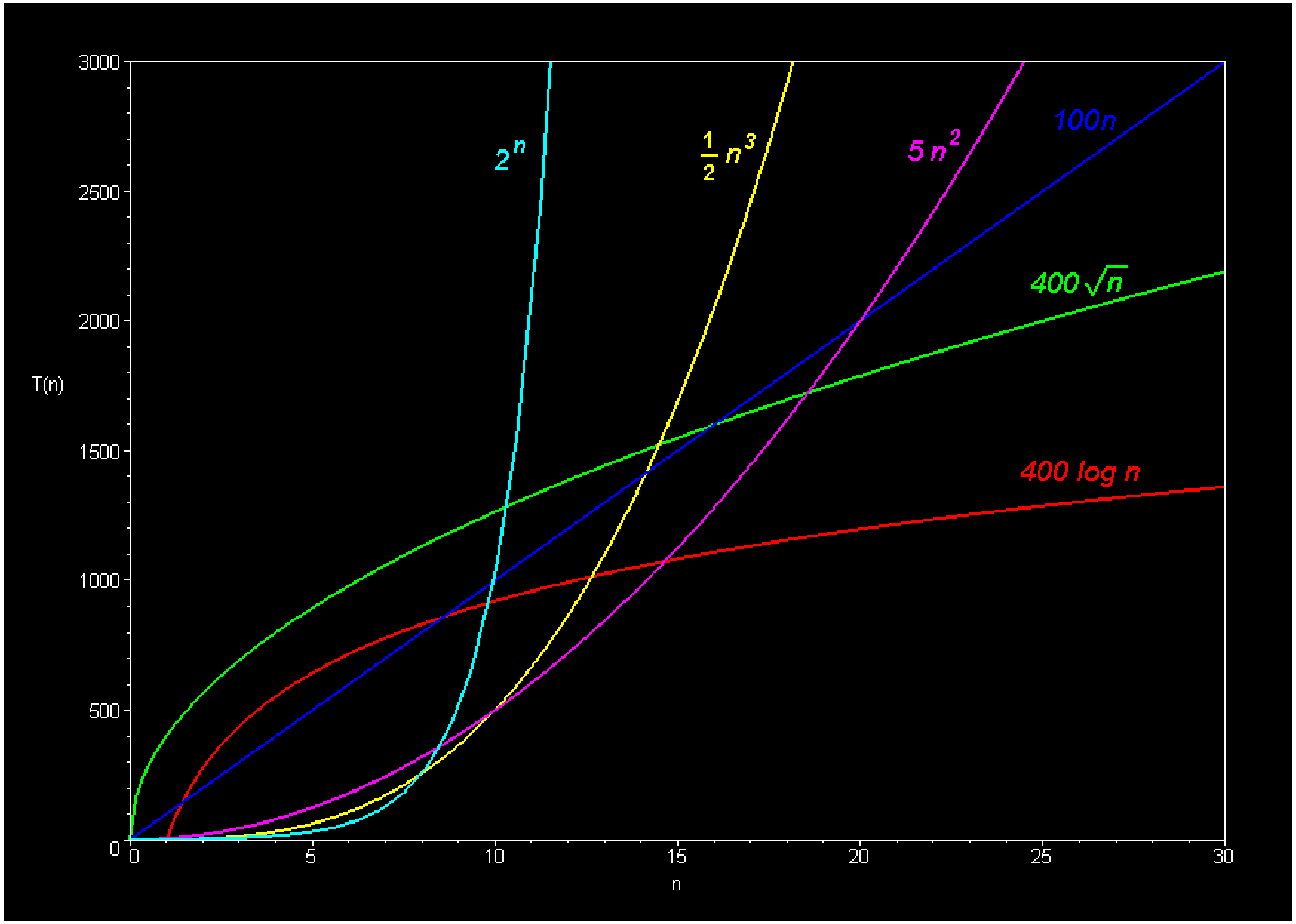
Asymptotic Growth Function

- In the previous example, the average and the worst case runtime would both double if the input size were doubled.
- The actual runtime would need to be measured on a specific platform, but we can characterize the runtime behavior to increase *linearly* with the input size.

Expressing Asymptotic Behavior

- Computer Scientists use the big-O notation to capture the essence of the worst case behavior of an algorithm or program.
- It is a function that describes the growth behavior of runtime or memory/storage requirements for a program.
- For the previous example, the program would have a time complexity of **O(n)**.

Program Complexity



Common Complexities

Constant run time independent of size of input	O(1)
Time increases linearly with size of input	O(n)
Time increases quadratically with size of input	$O(n^2)$
Time increases exponentially with size of input (extremely fast increase)	$O(2^n)$
Time increases logarithmically with size of input (very slow increase)	O(*log* n)
Time increases log-linear with size of input (a bit faster than linear)	O(n *log* n)

Calculate Complexity for an Algorithm

- Consider this simple function that sums a list of numeric objects:

```
# add the numbers in the vector
addNums <- function (v)  {
  l <- length(v)
  s <- 0
  for (i in 1:l) {
    s <- s + v[i]
  }
  return (s)
}

s <- addNums(c(3,5,7,1,8,2,3))
s
```

Count the Number of Steps

- To evaluate the running time of an algorithm, we will simply ask how many “steps” it takes.
- In this case, we can count the number of times it performs the addition.
- For a vector with *n* elements, it takes *n* steps, therefore this function has a time complexity of **O(n)**.

Measuring Performance

- Here's an actual measurement of the time in milliseconds using `system.time()`.

```
> system.time(s <- addNums(seq(from=1,to=1000000)))
   user  system elapsed
   0.64    0.00    0.64
> system.time(s <- addNums(seq(from=1,to=2000000)))
   user  system elapsed
   1.23    0.00    1.23
```

- Notice how a vector of twice the size takes about twice as long to be summed.

A Poor Implementation

- Novice programmers often write this code to add a vector of numbers.

```
# add the numbers in the vector
addNums <- function (v)
{
  s <- 0
  for (i in 1:length(v))
  {
    s <- s + v[i]
  }
  return (s)
}
```

- What is it's time complexity?

Time Complexity for Basic Operations

- Access *i*th element of a list: O(1).
- Search an list sequentially: O(n).
- Find an element in a sorted list. O(log n)
- Sorting:
  - Selection sort : O(n^2) in the best case
  - Insertion sort: O(n^2) on average
  - Quick sort: O(n *log* n) on average

Space Complexity

- Programs use memory and storage, so the computer scientist assesses a program’s use of that critical resource using big-O.
- For example, loading data into a vector has a space complexity of **O(n)** since you would need twice as much memory for a vector that is twice as big.

Summary

- In this lesson, you learned that:
  - the time and space complexity of a program must be calculated to understand its behavior as input size increases
  - complexity of time and space is given using big-O notation
  - the run-time of a function can be measured in R using `system.time()`

And I'll leave you with this final thought ;)

Summary, Review, & Questions…