Computational Concepts Toolbox

- Data type: values, literals, operations,
- Expressions, Call expression
- Variables
- Assignment Statement
- Sequences: tuple, list
- Dictionaries
- Data structures
- Tuple assignment
- Function Definition Statement
- Conditional Statement
- Iteration: list comp, for, while
- Lambda function expr.
- Higher Order Functions
  - as Values, Args, Results
- Higher order function patterns
  - Map, Filter, Reduce
  - Function factories
- Recursion
  - Linear, Tail, Tree
- Abstract Data Types
- Mutation
- Iterators and Generators
- Object Oriented Programming, Classes
- Exception
- Declarative Programming
- Distributed Computing

Recap: Complexity

- Example: Matrix Multiply
  - As a function of n?

```python
for i in 0 to n-1:
  for j in 0 to n-1:
    C[i][j] = 0
  for k in 0 to n-1:
    C[i][j] = C[i][j] + A[i][k]*B[k][j]
```

We say it is \( O(n^3) \) "big-O of \( n^3 \)" as an asymptotic upper bound

\[
time(n) < c \cdot n^3, \text{ for some suitably large constant } c \text{ for any instance of the inputs of size } n.
\]
A more subtle complexity example

- What is the “complexity” of finding the average number of factors of numbers up to n?

```python
def factors(n):
    return [x for x in range(2, max(n, ceil(sqrt(n)))) if n % x == 0]
def ave_factor(n):
    all_factors = map(factors, range(n))
    all_lens = map(len, all_factors)
    return sum(all_lens)/n
```

from timeit import default_timer as timer
def timeit(fun):
    """ Rtn timer for fun(i) in secs. """
    def timer_fun(i):
        start = timer()
        fun(i)
        end = timer()
        return (end-start)
    return timer_fun

How long does factors take?

In [9]: tbl = Table().with_column('n', np.arange(0.1, 1))
   tbl['factors'] = tbl.apply(factors, 'n')
tbl['n_factors'] = tbl.apply(len, 'factors')
tbl['secs'] = tbl.apply(timeit(factors), 'n')
tbl
Out[9]:
   n  n_factors  secs
0  0       0.876603e-06
1  0       2.40896e-06
2  0       1.34797e-06
3  0       3.49856e-06
4  [2]     2.74963e-06
5  0       2.43704e-06
6  [2, 3]  3.0190e-06
7  0       2.79e-06
8  [2, 4]  3.28396e-06
9  [3]     3.74601e-06
```

Big Data, Big Problems

- Performance terminology
  - the FLOP: FLoating point OPeration
  - “flops” = # FLOP/second is the standard metric for computing power

- Example: Global Climate Modeling
  - Divide the world into a grid (e.g. 10 km spacing)
  - Solve fluid dynamics equations for each point & minute
    » Requires about 100 Flops per grid point per minute
  - Weather Prediction (7 days in 24 hours):
    » 56 Gflops
  - Climate Prediction (50 years in 30 days):
    » 4.8 Tflops

- Perspective
  - Intel Core i7 980 XE Desktop Processor
    » ~100 Gflops
    » Climate Prediction would take ~5 years

What Can We Do? Use Many CPUs!

- Supercomputing – like those listed in top500.org
  - Multiple processors “all in one box / room” from one vendor that often communicate through shared memory
  - This is often where you find exotic architectures

- Distributed computing
  - Many separate computers (each with independent CPU, RAM, HD, NIC) that communicate through a network
    » Grids (heterogenous computers across Internet)
    » Clusters (mostly homogeneous computers all in one room)
    - Google uses commodity computers to exploit “knee in curve” price/ performance sweet spot
  - It’s about being able to solve “big” problems, not “small” problems faster
    » These problems can be data (mostly) or CPU intensive
Recap: Filter, Map, Reduce

- Functions as Data
- Higher-Order Functions
- Useful HOFs (you can build your own!)
  - `map function over list`
    - Report a new list, every element `e` of list becoming `function(e)`
  - `filter items such that predicate from list`
    - Create a new list, keeping only elements `e` of list if `predicate(e)`
  - `reduce with function over list`
    - Combine all the elements of list with `function(e)`

- Example:
  `filter -> map -> reduce`

Google’s MapReduce Simplified

- Filter: Chunk data and send to different CPUs.
- Map: Apply function to data chunks on different CPUs.
- Reduce: Combine results from different CPUs.
  - Reducer should be associative and commutative
- Imagine 10,000 machines ready to help you compute anything you could cast as a MapReduce problem!
  - This is the abstraction Google is famous for authoring
  - The system takes care of load balancing, dead machines, etc.

MapReduce: Advantages/Disadvantages

- Now it’s easy to program for many CPUs
  - Communication management effectively gone
  - Fault tolerance, monitoring
    » machine failures, suddenly-slow machines, etc are handled
  - Can be much easier to design and program!
  - Can cascade several (many?) MapReduce tasks

- But… it might restrict solvable problems
  - Might be hard to express problem in MapReduce
  - Data parallelism is key
    » Need to be able to break up a problem by data chunks

Apache Spark (from Berkeley)

- Data processing system that provides a simple interface to analytics on large data
- A Resilient Distributed Dataset (RDD) is a collection of values or key-value pairs
- Support the operations you are familiar with
  - Data-Parallel: map, filter, reduce
  - Database: join, union, intersect
  - OS: sort, distinct, count
- All of can be performed on RDDs that are partitioned across machines
Spark Execution Model

Processing is defined centrally and executed remotely
• A RDD is distributed over workers
• A driver program defines transformations and actions on RDDs
• A cluster manager assigns task to workers
• Workers perform computation, store data, & communicate with each other
• Final results communicate back to driver

Distributed Computing Challenges

• Communication is fundamental difficulty
  – Distributing data, updating shared resource, communicating results, handling failures
  – Machines have separate memories, so need network
  – Introduces inefficiencies: overhead, waiting, etc.
• Need to parallelize algorithms, data structures
  – Must look at problems from parallel standpoint
  – Best for problems whose compute times >> overhead

Speedup Issues: Amdahl’s Law

• Applications can almost never be completely parallelized; some serial code remains
  \[ \text{Speedup} = \frac{\text{Time}(1)}{\text{Time}(P)} \]
  \[ s \frac{1}{1 + (1-s) / P} \], and as \( P \to \infty \)
  \[ \leq 1 / s \]

Amdahl’s Law: Conclusion

• Computer Science View: Even if the parallel portion of your application speeds up perfectly, your performance will be limited by the sequential portion.
  🙁

• Data Science View: Often, as the data gets large, the work that can be parallelized grows faster than the size of the data.
  😊

  Fundamental Change in Perspective!
Summary: Data science

2.5 quintillion

https://www.youtube.com/watch?v=Tzmjbl-i4Y

Summary: Distributed Computing

- Parallelization can help speed up
- Bottleneck: dependencies in data, algorithm
- Requires rethinking the program

- Good luck with the project!
- See you next week for that final (fun) lecture!