Algorithms and Complexity

Amanda Robinson

About Me

Overview

- Algorithms
  - What is an algorithm?
  - Algorithm examples
- Complexity and running time
  - What is complexity?
  - How to calculate running time
  - Running time examples
  - Space vs. time
- Parallelization and MapReduce
  - MapReduce examples

What is an algorithm?

- Set of instructions to accomplish a task
- Slightly more formally, “any well-defined computational procedure that takes some value, or set of values, as **input** and produces some value, or set of values, as **output**.”[1]

1. Cormen, Leirson, Rivest, and Stein; Introduction to Algorithms; p. 5

Pseudocode

- Uses code-like notation to describe an algorithm or function at a high level
  - Don’t need to follow specific syntax of any language
  - Meant to be human-readable, not machine-readable
  - Commonly used in technical interviews, especially whiteboard sessions
- This lecture uses very Python-like pseudocode

```
# Comments are preceded by pound symbol
def my_function(input):  # def = definition
  # Blocks (for_each, function def.) denoted
  # by ‘:’ at beginning and by indentation
  for_each x in input: # Start for_each
    print x            # Inside for_each
  return result        # After for_each
```

Algorithm Example: Sum

```
def sum(arr):
  result = 0
  for_each x in arr:
    result = result + x
  return result
```

Algorithm Example: Insertion Sort

```
# 1-indexed: first element is arr[1]
def sort(arr):
  for j = 2 to length(arr):
    # Save the value at j
    key = arr[j]
    i = j - 1
    while i > 0 and arr[i] > key:
      # shift every item over 1 to make
      # room for key in its place
      arr[i + 1] = arr[i]
      i = i - 1
    arr[i + 1] = key
```

Complexity and Running Time

- ***Analyzing*** an algorithm: predicting resources needed
  - Usually, this means how much time will be required to complete the algorithm
  - Assumes a single processor, for now
- ***Running Time*** of an algorithm: how many steps are completed
  - Main question: as number of inputs N increases, how fast does Running Time increase?
  - Generally want to focus on worst-case

Complexity and Running Time

- ***Order of Growth*** or ***Rate of Growth*** describes about how fast the running time increases with increased size of input
  - Generally denoted with “Big-O” Notation
  - Sometimes will use θ
  - Something that does not grow with size of input: O(1) or θ(1), also know as Constant time
  - Linear growth: O(N)
    - N represents the number of inputs
  - Quadratic: $O(N^2)$
  - Logarithmic: O(log(N))
  - etc.

Complexity and Running Time

- Calculating Running Time
  - Loops (`for`, `while`, etc.) tend to be multiplicative:

```
for x in 1 to N:
  print x
```

    print x runs in constant time, but is run N times, for O(1 * N) = O(N) running time
  - Nested loops multiply further:

```
for x in 1 to N:
  for y in 1 to N:
    print x, y
```

    print x is still constant time, but runs N * N times, for $O(N^2)$ running time
  - Non-nested loops are simply additive:

```
for x in 1 to N:
  print x
for y in 1 to N:
  print y
```

    —> O(N) for first loop + O(N) for second loop = O(2N) = O(N)
  - NOTE: Do not care about multipliers (except when comparing two different algorithms): O(10N) = O(N)
  - NOTE: When there are multiple inputs, running time may depend on all of them: O(NM) for two arrays of sizes N and M

Running Time Example: Sum

```
def sum(arr):
  result = 0                    # O(1)
  for_each x in arr:            # O(N)
    result = result + x         # O(1)
  return result                 # O(1)

Running time = O(1 + (N * 1) + 1)
             = O(1 + N + 1)
             = O(N)
```

Running Time Example: Insertion Sort

```
def sort(arr):
  for j = 2 to length(arr):         # O(N - 1)
    key = arr[j]                    # O(1)
    i = j - 1                       # O(1)
    while i > 0 and arr[i] > key:   # O(N - 1)
      arr[i + 1] = arr[i]           # O(1)
      i = i - 1                     # O(1)
    arr[i + 1] = key                # O(1)

RT = O((N - 1) * (1 + 1 + (N - 1) * 1 + 1)
   = O((N - 1) * (N - 1))
   = O(N * N)
   = O(N^2)
```

Running Time Example: Sorting Phone Numbers

Special case: When the problem space is known and relatively small (max value < 10,000,000), and there is enough space available, some things can be accomplished much faster.

```
def sort(arr):
  numbers = new Array(10000000, False)
  for j = 1 to length(arr):
    numbers[ arr[j] ] = True
  i = 1
  for j = 1 to length(numbers):
    if numbers[j] == True:
      arr[i] = j
      i = i + 1
```

Running Time Example: Sorting Phone Numbers

Special case: When the problem space is known and relatively small (max value < 10,000,000), and there is enough space available, some things can be accomplished much faster.

```
def sort(arr):
  numbers = new Array(10000000, False) # O(1)
  for j = 1 to length(arr):            # O(N)
    numbers[ arr[j] ] = True           # O(1)
  i = 1                                # O(1)
  for j = 1 to length(numbers):        # O(10000000)
    if numbers[j] == True:             # O(1)
      arr[i] = j                       # O(1)
      i = i + 1                        # O(1)

RT = O(1 + (N * 1) + 1 + (10000000 * (1 + 1 + 1)))
   = O((N * 1) + 10000000)
   = O(N * 1)
   = O(N)
```

Time vs. Space

- For certain problems, the amount of time it takes to solve them can be reduced by increasing the amount of space (memory) used.
- Similarly, again for certain problems, the time can be reduced by using multiple processors via ***parallelization***.
    - Sum: Would work
        - Split array in half, have each processor calculate the sum of 1/2 of the array, then add the two numbers together, to take about 1/2 the total time.
        - Split into S segments, and use S processors, to take 1/S total time.
    - Insertion Sort: Would not work
        - Cannot have multiple processors updating the same array at once)
- NOTE: Parallelizing a function does not change its running time—a $O(N^2)$ algorithm will still be a $O(N^2)$ algorithm. It simply lowers the total time taken to complete calculations.
- When limited to your laptop (2-4 processors), this has relatively little effect for a very large problem. But what if you had 1000 processors?

Parallelism and MapReduce

- Parallelism, or Parallel Computing, uses multiple processors to process large datasets
  - Single computer: most personal computers today
  - Multiple computers: datacenters, etc.
- MapReduce is one popular parallel computing method
  - 2 steps: Map step and Reduce step
  - Map: Splits the dataset into smaller pieces for parallel processing
  - Reduce: Collects the results of the mapped processes back together into a single result

MapReduce Example: Sum

- Input: [1, 3, 6, 23, 85, 53, 75, 2]
- Map:
    - Split into 2 arrays:
        - [1, 3, 6, 23] and [85, 53, 75, 2]
    - Sum each array:
        - 1 + 3 + 6 + 23 = 33
        - 85 + 53 + 75 + 2 = 215
- Reduce:
    - Sum the two results:
        - 33 + 215 = 248

MapReduce Anti-Example: Insertion Sort

- Input: [1, 3, 6, 23, 85, 53, 75, 2]
- Map:
  - Split into 2 arrays:
    - [1, 3, 6, 23] and [85, 53, 75, 2]
  - Sort each array:
    - [1, 3, 6, 23] —> [1, 3, 6, 23]
    - [85, 53, 75, 2] —> [2, 53, 75, 85]
- Reduce:
  - If one array is insertion-sorted into the second, there is no time savings
  - HOWEVER: Could create a new, sorted array from the two in N time:
    - Look at the first element of each array
    - Take the lowest value out of that array and append to new array
    - This reduce step runs in O(N) time
    - More like Merge Sort (https://en.wikipedia.org/wiki/Merge_sort)

MapReduce Example: Stepwise Selection

- For each X in inputs, fit the model; then, take the best fit

- Can do this step on multiple processors (up to number of inputs), then take the best one, instead of fitting for each input in serial

- Map step: fit model for given X from inputs, calculate AIC for that X

- Reduce step: take the model with the best AIC