Data structures in Java

Session 3

Instructor: Bert Huang

http://www1.cs.columbia.edu/~bert/courses/3134

Announcements

- HW1: **Collection** test function on homepage. Small typo fixed from Saturday (redownload if necessary)
- Due on 9/22 by class time; that is in a little less than 7 days

Review

- Java review
  - Syntax and Java paradigms
  - Classes, encapsulation, hierarchy
- Math review
  - Exponents, logarithms, summations

Today's Plan

- Algorithm Analysis
- Big-Oh notation

Big-Oh Notation

- We adopt special notation to define **upper bounds** and **lower bounds** on functions
- In CS, usually the functions we are bounding are running times, memory requirements.
- We will refer to the running time as $T(N)$

Definitions

- For *N* greater than some constant, we have the following definitions:

$$
\begin{aligned}
T(N) &= O(f(N)) \leftarrow T(N) \leq cf(N) \\
T(N) &= \Omega(g(N)) \leftarrow T(N) \geq cf(N) \\
T(N) &= \Theta(h(N)) \leftarrow \begin{array}{c} T(N) = O(h(N)), \\ T(N) = \Omega(h(N)) \end{array}
\end{aligned}
$$

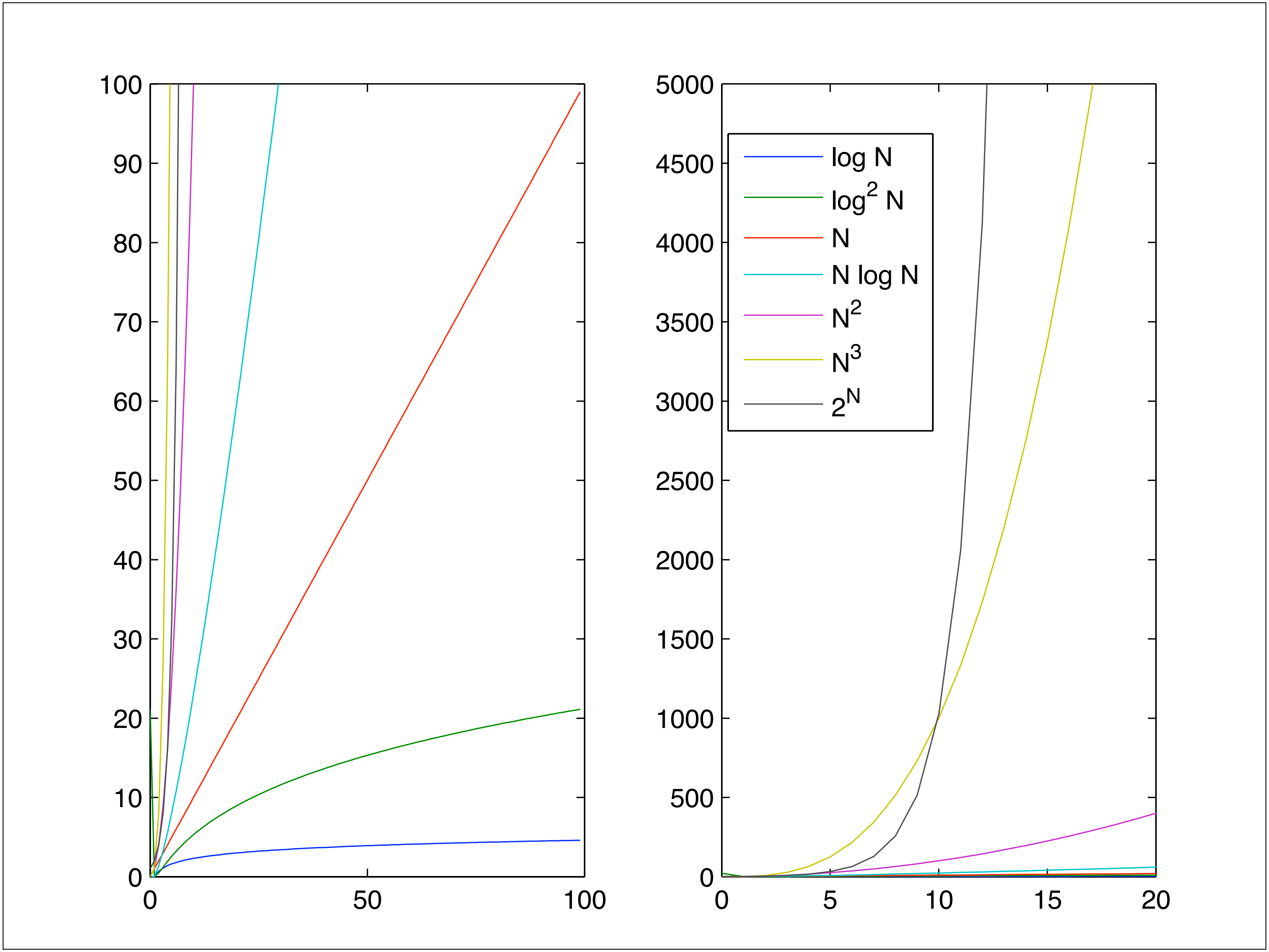
- There exists some constant c such that cf(N) bounds T(N)

Definitions

- Alternately, O(f(N)) can be thought of as meaning

$$T(N) = O(f(N)) \leftarrow \lim_{N\to\infty} f(N) \geq \lim_{N\to\infty} T(N)$$

- Big-Oh notation is also referred to as **asymptotic** analysis, for this reason.



100
90
80
70
60
50
40
30
20
10
0
0
50
100
5000
4500
4000
3500
3000
2500
2000
1500
1000
500
0
0
5
10
15
20
log N
log^2 N
N
N log N
N^2
N^3
2^N

Comparing Growth Rates

$T_1(N) = O(f(N))$ and $T_2(N) = O(g(N))$

then

$$\begin{array}{ll} (a) & T_1(N) + T_2(N) = O(f(N) + g(N)) \\ (b) & T_1(N)T_2(N) = O(f(N)g(N)) \end{array}$$

- If you have to, use l'Hôpital's rule

$$\lim_{N\to\infty} f(N)/g(N) = \lim_{N\to\infty} f'(N)/g'(N)$$

Example: Maximum Subsequence

- Given a sequence of integers (possibly negative), find the subsequence whose sum is the maximum

-2	11	-4	13	-5	-2

- We'll look at three algorithms: slow, medium, fast

Naïve Algorithm

```
for i=1 to N {
    for j=i to N {
        sum = 0
        for k=i to j
            sum = sum+A[k]
        if (sum > maxSum)
            maxSum = sum
    }
}
```

$$T(N) = \sum_{i=1}^{N} \sum_{j=i}^{N} \sum_{k=i}^{j} 1$$

-2	11	-4	13	-5	-2

Naïve Algorithm

$$T(N) = \sum_{i=1}^{N} \sum_{j=i}^{N} \sum_{k=i}^{j} 1$$

$$T(N) = \sum_{i=1}^{N} \sum_{j=i}^{N} j - i + 1$$

$$T(N) = \sum_{i=1}^{N} \frac{(N-i+2)(N-i+1)}{2}$$

$$T(N) = \frac{N^3 + 3N^2 + 2N}{6}$$

Better Algorithm

- for i=1 to N {

```
    sum = 0
    for j=i to N {
        sum += A[j]
        if (sum>maxSum)
            maxSum = sum
    }
}
```

$$\sum_{i=1}^{N}\sum_{j=i}^{N} 1$$

$$\sum_{i=1}^{N} N - i + 1$$

-2	11	-4	13	-5	-2

Better Algorithm

$$\sum_{i=1}^{N} N - i + 1$$

$$\sum_{i=1}^{N} N - \sum_{i=1}^{N} i + \sum_{i=1}^{N} 1$$

$$N^2 - \frac{N(N+1)}{2} + N$$

$$\frac{N^2 + N}{2}$$

Best Algorithm?

- ```
  for j=1 to N {
      sum += A[j]
      if (sum > maxSum)
          maxSum = sum
      else if (sum < 0)
          sum = 0
  }
  ```

-2	11	-4	13	-5	-2
  ```

Logarithmic Running Time

- Rule of thumb: If it takes constant time to reduce the problem size by a fraction, usually O(log N)
- Given integer **x** and array **A**, which is sorted, find the index of **A** that is equal to **x** (or return failure)
- Use **Binary Search**

Binary Search

- (very simplified for readability)
- BinarySearch(x,A)

```
mid = median
if ( x > A[mid])
    BinarySearch(x, A[mid to end])
else
    BinarySearch(x, A[0 to mid])
```

-4	-2	0	3	4	6	8	12	16	18	19	20	21	22	30	99

Recurrences

- Sometimes the rule of thumb is insufficient
- Instead, write the running time as a recursive function:

$$\begin{aligned} T(N) &= c + T(N/2) \\ T(N/2) &= c + T(N/4) \\ T(N/4) &= c + T(N/8) \end{aligned}$$

Recurrences

- Sometimes the rule of thumb is insufficient
- Instead, write the running time as a recursive function:

$$T(N) = c + c + T(N/4)$$

$$T(N/4) = c + T(N/8)$$

Recurrences

- Sometimes the rule of thumb is insufficient
- Instead, write the running time as a recursive function:
  $T(N) = c + c + c + T(N/8)$

Recurrences

- Sometimes the rule of thumb is insufficient
- Instead, write the running time as a recursive function:
  $T(N) = c + c + c + T(N/8)$
- How many times can this repeat?

$$T(N) = \sum_{i=1}^{\log N} c$$

$$T(N) = \Theta(\log N)$$

Empirical Running Time

- Timing algorithms can usually give you a good idea if your big-oh analysis is accurate
- Average running times can be very different: e.g., Quicksort runs in N log N, but is worst case quadratic
- Best case inputs can occur too, confusing your analysis

Reading

- Today’s class covered Weiss Ch. 2
- Thursday’s class will cover Weiss Ch. 3