Data Abstraction
Algorithm Analysis (4.2, 4.3)

Janyl Jumadinova

October 19-23, 2020
Analysis of Algorithms

Seven functions that often appear in algorithm analysis:

1. Constant $\approx 1$
2. Logarithmic $\approx \log n$
3. Linear $\approx n$
4. $N - \log N \approx n \log n$
5. Quadratic $\approx n^2$
6. Cubic $\approx n^3$
7. Exponential $\approx 2^n$
In a log-log chart, the slope of the line corresponds to the **growth rate**.
Counting Primitive Operations

By inspecting the pseudocode, we can determine the maximum number of primitive operations executed by an algorithm, as a function of the input size.
Counting Primitive Operations

By inspecting the pseudocode, we can determine the maximum number of primitive operations executed by an algorithm, as a function of the input size.

```java
/** Returns the maximum value of a nonempty array of numbers. */
public static double arrayMax(double[] data) {
    int n = data.length;
    double currentMax = data[0];  // assume first entry is biggest (for now)
    for (int j=1; j < n; j++) {    // consider all other entries
        if (data[j] > currentMax)  // if data[j] is biggest thus far...
            currentMax = data[j];  // record it as the current max
    }
    return currentMax;
}
```
Counting Primitive Operations

By inspecting the pseudocode, we can determine the maximum number of primitive operations executed by an algorithm, as a function of the input size.

```java
1  /** Returns the maximum value of a nonempty array of numbers. */
2  public static double arrayMax(double[] data) {
3      int n = data.length;
4      double currentMax = data[0];               // assume first entry is biggest (for now)
5      for (int j=1; j < n; j++)                 // consider all other entries
6         if (data[j] > currentMax)             // if data[j] is biggest thus far...
7             currentMax = data[j];             // record it as the current max
8      return currentMax;                      //
9  }
```

Step 3: 2 ops, 4: 2 ops, 5: 2n ops, 6: 2n ops, 7: 0 to n ops, 8: 1 op
Estimating Running Time

2 + 2 + 2n + 2n + 0 ... n + 1

Then, algorithm `arrayMax` executes:

- $5n + 5$ primitive operations in the worst case
- $4n + 5$ in the best case
Let $a = \text{Time taken by the fastest primitive operation.}$
Estimating Running Time

- Let \( a \) = Time taken by the fastest primitive operation.
- Let \( b \) = Time taken by the slowest primitive operation.
Estimating Running Time

- Let $a$ = Time taken by the fastest primitive operation.
- Let $b$ = Time taken by the slowest primitive operation.
- Let $T(n)$ be worst-case time of `arrayMax`.

Then,

$$a(4n + 5) \leq T(n) \leq b(5n + 5)$$

Hence, the running time $T(n)$ is bounded by two linear functions.
Estimating Running Time

- Let $a =$ Time taken by the fastest primitive operation.
- Let $b =$ Time taken by the slowest primitive operation.
- Let $T(n)$ be worst-case time of $\text{arrayMax}$.
- Then, $a(4n + 5) \leq T(n) \leq b(5n + 5)$
Let $a =$ Time taken by the fastest primitive operation.

Let $b =$ Time taken by the slowest primitive operation.

Let $T(n)$ be worst-case time of `arrayMax`.

Then, $a(4n + 5) \leq T(n) \leq b(5n + 5)$

Hence, the running time $T(n)$ is bounded by two linear functions.
Changing the hardware/ software environment

- Affects \( T(n) \) by a constant factor, but
- Does not alter the growth rate of \( T(n) \)

The linear growth rate of the running time \( T(n) \) is an intrinsic property of algorithm arrayMax.
Why Growth Rate Matters

<table>
<thead>
<tr>
<th>if runtime is...</th>
<th>time for n + 1</th>
<th>time for 2n</th>
<th>time for 4n</th>
</tr>
</thead>
<tbody>
<tr>
<td>$c \lg n$</td>
<td>$c \lg (n + 1)$</td>
<td>$c (\lg n + 1)$</td>
<td>$c(\lg n + 2)$</td>
</tr>
<tr>
<td>$cn$</td>
<td>$c (n + 1)$</td>
<td>$2cn$</td>
<td>$4cn$</td>
</tr>
<tr>
<td>$cn \lg n$</td>
<td>$\sim cn \lg n$</td>
<td>$2cn \lg n + 2cn$</td>
<td>$4cn \lg n + 4cn$</td>
</tr>
<tr>
<td>$cn^2 + cn$</td>
<td>$\sim cn^2 + 2cn$</td>
<td><strong>$4cn^2$</strong></td>
<td>$16cn^2$</td>
</tr>
<tr>
<td>$cn^3 + 3cn^2$</td>
<td>$\sim cn^3 + 3cn^2$</td>
<td>$8cn^3$</td>
<td>$64cn^3$</td>
</tr>
<tr>
<td>$c2^n$</td>
<td>$c2^{n+1}$</td>
<td>$c2^{2n}$</td>
<td>$c2^{4n}$</td>
</tr>
</tbody>
</table>

runtime quadruples when problem size doubles
Comparison of Two Algorithms

- **InsertionSort** is \( \frac{n^2}{4} \)
- **MergeSort** is \( 2n\log n \)

![Graph of Insertion Sort vs Merge Sort](image)
Comparison of Two Algorithms

- InsertionSort is $\frac{n^2}{4}$
- MergeSort is $2n \log n$

To sort a million items?
- InsertionSort takes roughly 70 hours
- MergeSort takes roughly
Comparison of Two Algorithms

- InsertionSort is $\frac{n^2}{4}$
- MergeSort is $2n\log n$

To sort a million items?

- InsertionSort takes roughly **70 hours**
- MergeSort takes roughly **40 seconds**
Comparison of Two Algorithms

- InsertionSort is $\frac{n^2}{4}$
- MergeSort is $2n \log n$

To sort a million items?

- InsertionSort takes roughly 70 hours
- MergeSort takes roughly 40 seconds

This is a slow machine, but if 100 x as fast then it is 40 minutes versus less than 0.5 seconds.
Given functions $f(n)$ and $g(n)$, we say that $f(n)$ is $O(g(n))$ if there are positive constants $c$ and $n_0$ such that $f(n) \leq cg(n)$ for $n \geq n_0$.
Big-Oh Examples

- $7n - 2$ is $O(n)$
Big-Oh Examples

- \(7n - 2\) is \(O(n)\) need \(c > 0\) and \(n_0 \geq 1\) such that \(7n - 2 \leq cn\) for \(n \geq n_0\)
  this is true for \(c = 7\) and \(n_0 = 1\)
Big-Oh Examples

- $7n - 2$ is $O(n)$ need $c > 0$ and $n_0 \geq 1$ such that $7n - 2 \leq cn$ for $n \geq n_0$
  - this is true for $c = 7$ and $n_0 = 1$
- $3n^3 + 20n^2 + 5$ is $O(n^3)$
Big-Oh Examples

- $7n - 2$ is $O(n)$ need $c > 0$ and $n_0 \geq 1$ such that $7n - 2 \leq cn$ for $n \geq n_0$
  this is true for $c = 7$ and $n_0 = 1$

- $3n^3 + 20n^2 + 5$ is $O(n^3)$ need $c > 0$ and $n_0 \geq 1$ such that $3n^3 + 20n^2 + 5 \leq cn^3$ for $n \geq n_0$
  this is true for $c = 4$ and $n_0 = 21$
Big-Oh Examples

- $7n - 2$ is $O(n)$ need $c > 0$ and $n_0 \geq 1$ such that $7n - 2 \leq cn$ for $n \geq n_0$
  
  this is true for $c = 7$ and $n_0 = 1$

- $3n^3 + 20n^2 + 5$ is $O(n^3)$ need $c > 0$ and $n_0 \geq 1$ such that
  
  $3n^3 + 20n^2 + 5 \leq cn^3$ for $n \geq n_0$

  this is true for $c = 4$ and $n_0 = 21$

- $3\log n + 5$ is $O(\log n)$
Big-Oh Examples

- $7n - 2$ is $O(n)$ need $c > 0$ and $n_0 \geq 1$ such that $7n - 2 \leq cn$ for $n \geq n_0$
  this is true for $c = 7$ and $n_0 = 1$

- $3n^3 + 20n^2 + 5$ is $O(n^3)$ need $c > 0$ and $n_0 \geq 1$ such that $3n^3 + 20n^2 + 5 \leq cn^3$ for $n \geq n_0$
  this is true for $c = 4$ and $n_0 = 21$

- $3\log n + 5$ is $O(\log n)$ need $c > 0$ and $n_0 \geq 1$ such that $3\log n + 5 \leq c\log n$ for $n \geq n_0$
  this is true for $c = 8$ and $n_0 = 2$
Big-Oh and Growth Rate

The big-Oh notation gives an upper bound on the growth rate of a function.

The statement "$f(n)$ is $O(g(n))$" means that the growth rate of $f(n)$ is no more than the growth rate of $g(n)$.
The asymptotic analysis of an algorithm determines the running time in big-Oh notation.

To perform the asymptotic analysis:

1. We find the worst-case number of primitive operations executed as a function of the input size.
The asymptotic analysis of an algorithm determines the running time in big-Oh notation.

To perform the asymptotic analysis:

1. We find the worst-case number of primitive operations executed as a function of the input size.
2. We express this function with big-Oh notation.
The asymptotic analysis of an algorithm determines the running time in big-Oh notation.

To perform the asymptotic analysis:
1. We find the worst-case number of primitive operations executed as a function of the input size.
2. We express this function with big-Oh notation.

Example: We say that algorithm `arrayMax` “runs in O(n) time” Since constant factors and lower-order terms are eventually dropped anyhow, we can disregard them when counting primitive operations.