What are we going to learn?

- Need to say that some algorithms are “better” than others
- Criteria for evaluation
  - Structure of programs (simplicity, elegance, etc.)
  - Running time
  - Memory Space

Overview

- Running Time
- Pseudo-Code
- Analysis of algorithms
- Asymptotic notations
- Asymptotic analysis

Algorithm, and Inputs

- The running time of algorithms typically depends on the input set, and its size \(n\).
- We describe it using functions of \(n\).
Average Case vs. Worst Case

- The average case behavior is harder to analyze because you need to know a probability distribution of the input.
- In certain apps (air traffic control, weapon systems, etc.), knowing the worst case time is important.

Measuring the running time

- How should we measure the running time of an algorithm?

Experimental Studies

- Run the program with various data sets in a computer, and measure the wall clock time.

General Methodology

- Independent of implementation, hardware and software environments
- Actual elapsed time depends on
  - hardware, software (os), compiler.
- Use high-level description of the algorithm instead of one of its implementations
General Methodology

- Worry about order of magnitude
  - count steps (don't worry about amount of time each step takes)
  - ignore multiplicative constants
- Take into account all possible inputs.
  - Worst cast analysis

Pseudo-Code

- Pseudo-code is a description of an algorithm for human-eyes only (mix of English and programming languages).
- Example: finding the maximum element of an array.

```
Algorithm arrayMax(A, n):
  Input: An array A storing n integers.
  Output: The maximum element in A.
  currentMax ← A[0]
  for i ← 1 to n − 1 do
    if currentMax < A[i] then
      currentMax ← A[i]
  return currentMax
```

How to count steps

- Comments, declarative statements (0)
- expressions and assignments (1)
  - except for function calls
  - cost for fn needs to be counted separately and added to the cost for the calling statement.

How to count steps iteration

- Iteration statements -- for, while
  - expression + count the number of times that the body is executed, and then multiply by the cost of body
  - while <expr> do
    <body of while>
How to count steps

- Running time of worst case + expression.

```java
switch <expr>
    case cond1: <statement1>
    case cond2: <statement2>
    ....
```

Example

**Algorithm arrayMax(A, n):**

- **Input:** An array A storing n integers.
- **Output:** The maximum element in A.

```java
currentMax ← A[0]
for i ← 1 to n - 1 do
    if currentMax < A[i] then
        currentMax ← A[i]
return currentMax
```

Example of analysis

**Algorithm prefixAverages(X):**

- **Input:** An n-element array X of numbers.
- **Output:** An n-element array A of numbers such that A[i] is the average of elements X[0], ..., X[i].

Let A be an array of n numbers.

```java
for i ← 0 to n - 1 do
    a ← 0
    for j ← 0 to i do
        a ← a + X[j]
    A[i] ← a/(i + 1)
return array A
```

Another Example

```java
Result <-0; m <-1;
for i<-1 to n
    m <- m*2;
for j<- 1 to m do
    result <- result + i*m*j
```
**Estimating Running Time**

- Algorithm `arrayMax` executes $3n - 1$ primitive operations in the worst case.
- Define
  - $a$: Time taken by the fastest primitive operation
  - $b$: Time taken by the slowest primitive operation
- Let $T(n)$ be the actual worst-case running time of `arrayMax`. We have
  $$a(3n - 1) \leq T(n) \leq b(3n - 1)$$
- Hence, the running time $T(n)$ is bounded by two linear functions.

**Growth Rate of Running Time**

- 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`.

---

**Growth Rates**

- Growth rates of functions:
  - Linear $\approx n$
  - Quadratic $\approx n^2$
  - Cubic $\approx n^3$
- In a log-log chart, the slope of the line corresponds to the growth rate of the function.

**Constant Factors**

- The growth rate is not affected by
  - constant factors of
  - lower-order terms
- Examples
  - $10^9n + 10^n$ is a linear function
  - $10^9n^2 + 10^n$ is a quadratic function.
Big-Oh Notation

- 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 \( n \geq n_0 \).

- Example: \( 2n + 10 \) is \( O(n) \)
  - \( 2n + 10 \leq cn \)
  - \( (c - 2) \cdot n \geq 10 \)
  - \( c \geq 10 \) and \( n_0 = 10 \)
  - Or Pick \( c = 4 \) and \( n_0 = 5 \)
  - Just prove the existence of \( c \) and \( n_0 \).

Big-Oh Notation (cont.)

- Example: the function \( n^2 \) is not \( O(n) \)
  - \( n^2 \leq cn \)
  - \( n \leq c \)
  - The above inequality cannot be satisfied since \( c \) must be a constant.

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 \( g(n) \).
- We can use the big-Oh notation to rank functions according to their growth rate.

<table>
<thead>
<tr>
<th>( f(n) ) grows more</th>
<th>( f(n) ) is ( O(g(n)) )</th>
<th>( g(n) ) is ( O(f(n)) )</th>
</tr>
</thead>
<tbody>
<tr>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>No</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>Same growth</td>
<td>Yes</td>
<td>Yes</td>
</tr>
</tbody>
</table>

Classes of Functions

- Let \( \{g(n)\} \) denote the class (set) of functions that are \( O(g(n)) \).
- We have \( \{n\} \subseteq \{n^2\} \subseteq \{n^3\} \subseteq \{n^4\} \subseteq \ldots \)
- where the containment is strict.
**Big-Oh Rules (of Thumb)**

- If is \( f(n) \) a polynomial of degree \( d \), then \( f(n) \) is \( O(n^d) \), i.e.,
  1. Drop lower-order terms
  2. Drop constant factors
- Use the smallest possible class of functions
  - Say "2n is \( O(n) \)" instead of "2n is \( O(n^2) \)"
  - But it is true that 2n is \( O(n^2) \)
- Use the simplest expression of the class
  - Say "3n + 5 is \( O(n) \)" instead of "3n + 5 is \( O(3n) \)"
  - But it is true that 3n + 5 is \( O(3n) \)

**Asymptotic Algorithm Analysis**

- 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 determine that algorithm `arrayMax` executes at most \( 3n - 1 \) primitive operations
  - We say that algorithm `arrayMax` "runs in \( O(n) \) time"
- Since constant factors and lower-order terms are eventually dropped anyhow in big-Oh, we can disregard them when counting primitive operations

**Caution!**

- Beware of very large constant factors. An algorithm running in time \( 1,000,000n \) is still \( O(n) \), but might be less efficient on your "everyday" data set than one running in time \( 2n^2 \), which is \( O(n^2) \).