What is an Algorithm?

• An **algorithm** is a **well-defined computational procedure** that transforms some input value (or set of values) into an output value (or set of values)

• A sequence of **computational steps** (step-by-step instructions) that tell a **computing agent** how to solve a **computational problem** using **finite resources**

• Resources
  • CPU Cycles
  • Memory
  • Best case / Worst case / Average case
Pseudocode

- English-like description of the operation of an algorithm
  - Including sequential, conditional and iterative operations
- No rigid syntax rules. Should be clear, complete, and organized.
- Example: Find Maximum
  - Input: A list of positive integers
  - Output: The largest number in the list

```
largest = 0
for each index in list:
    if list[index] > largest
        largest = list[index]

set largest to 0
set current-number to first element in list
while there are more numbers in list:
    if current-number > largest
        set largest to current-number
    set current-number to next element in list
```

Either of these is OK for this class!
Choosing an Algorithm

• Often many algorithms exist to solve the same problem
  • How to choose "the best"?
  • What does "the best" even mean?
• Want to find the most efficient algorithm
• What do we mean by efficiency?
  • Time (CPU cycles)
    • We'll focus on this (but others are [increasingly?] important too)
  • Space (memory)
  • Work
  • Power
Algorithm Analysis

- **Analyzing an algorithm** means predicting the resources it will require
  - Again, "resources" can mean several things
  - In this class, we mean *computation time*

- Assume 1 processor executing instructions one by one, with no concurrent operations, and constant execution time for all instructions and memory accesses
  - *Count the number of basic operations*: addition, subtraction, multiplication, division, comparison, etc.
Analyzing Computation Time

- The number of basic operations performed by an algorithm is a function of:
  - Problem (input) size
  - Sometimes also the nature of the input data
    - E.g., for a sort algorithm, is the input data already mostly sorted?
- So if it's input-dependent, how can we generically discuss an algorithm's computation time?
  - Best-case runtime
  - Average runtime
  - **Worst-case runtime**
    - Most common, what we'll use in this class
Algorithm Complexity/Efficiency

• To classify the efficiency of an algorithm:
  • Express the algorithm's computation time (# of basic operations) as a function of its input size
    • E.g.: \(aN + b\)
      where \(N\) is the number of inputs, \(a\) & \(b\) are constants
  • Determine the mathematical classification of the function:

<table>
<thead>
<tr>
<th>Complexity</th>
<th>Function</th>
<th>Big-Oh Notation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Constant</td>
<td>(f(N) = b)</td>
<td>(O(1))</td>
</tr>
<tr>
<td>Logarithmic</td>
<td>(f(N) = \log(N))</td>
<td>(O(\log N))</td>
</tr>
<tr>
<td>Linear</td>
<td>(f(N) = aN + b)</td>
<td>(O(N))</td>
</tr>
<tr>
<td>Linearithmic</td>
<td>(f(N) = N \times \log(N))</td>
<td>(O(N \log N))</td>
</tr>
<tr>
<td>Quadratic</td>
<td>(f(N) = aN^2 + bN + c)</td>
<td>(O(N^2))</td>
</tr>
<tr>
<td>Exponential</td>
<td>(b^N)</td>
<td>(O(2^N))</td>
</tr>
</tbody>
</table>

*"Big-Oh notation"*

Expresses upper bound on algorithm's time complexity

Ignores all but dominant term, drops constants
Comparing Function Growth

- $C=3$
- $\log(N)$
- $N$
- $N^2$
- $2^N$
- $N^3$
- $N!$

Values:
- 1
- 2
- 4
- 8
- 16
- 32
- 64
- 128
- 256
- 512
- 1024
- 2048
- 4096
- 8192
- 16384
- 32768
- 65536

Graph shows the comparison of growth rates for each function.
An Example...

- **Problem Statement:**
  
  Sum the numbers from 1 to N

- **Algorithm #1**

- **Analysis**

  - How many basic operations do we do?
    - An assign to initialize sum
    - Every loop iteration:
      - An addition operation
      - An assignment

  

  ![Graph showing time as a linear function of the problem size.](image)

  Time is a **linear** function of the problem size: \( O(N) \)
An Example...

- **Problem Statement:**
  Sum the numbers from 1 to N

- **Algorithm #2**
  - Consider N = 100
  - Key insight: numbers can be grouped as follows:
    1 + 100 = 101
    2 + 99 = 101
    ...
    50 + 51 = 101
  - Generalize the formula: \( \frac{N}{2} \times (N + 1) \)

- **Analysis:** Count the operations
  
  Time requirement is constant: \( O(1) \)
Sequential Search

• Suppose you want to find a Texas State student's name by looking up their NetID
  • Campus directory contains NetIDs, names, majors, etc.
  • Input: A NetID. Output: Student's name.
  • Assume N entries in campus directory

```python
found = false
i = 0
while !found and i < N
    if NetID of directory[i] matches search NetID
        found = true
    else i++
if !found
    NetID not in directory
else
    return name of directory[i]
```
Sequential Search Time Requirements

• **Best case** (minimum work):
  • NetID match in directory[0] (first entry)
  • One loop iteration

• **Worst case** (maximum work):
  • NetID match in directory[N-1] (last entry)
  • N loop iterations

• **Average case** (expected work):
  • NetID match in directory[N/2] (middle entry)
  • N/2 loop iterations

Worst and average case are both $O(N)$
So Is That Fast Enough?

- **Assume:**
  - \( N = 150,000 \) students registered during the past 10 years
  - Say your computer can do 50,000 of the loop iterations per second...

- **Average case:**
  \[
  \frac{150,000 \text{ iterations}}{2} \times \frac{1 \text{ second}}{50,000 \text{ iterations}} = 1.5 \text{ seconds}
  \]

- **Worst case:**
  \[
  \frac{150,000 \text{ iterations}}{2} \times \frac{1 \text{ second}}{50,000 \text{ iterations}} = 3.0 \text{ seconds}
  \]

What if you had to search the IRS database? (US population = 318 million)
Algorithm Design

- Designing algorithms to efficiently (or even feasibly) solve problems is the challenge of computer science
  - Code is a tool to implement the algorithms...
  - ...but programming is not (by itself) computer science!

- Algorithm design is not always easy
  - Example: The Traveling Salesman Problem
    - A salesperson needs to visit N cities and then return to his/her origin city. What is the shortest possible route that visits each city exactly once and then returns to the start city?
    - Applications: logistics, wire routing, genome analysis, ...
Traveling Salesman Problem (TSP)

- Brute-force
  - Try all permutations (city orderings, or *tours*) and compute which is cheapest (lowest total distance traveled)
- Guaranteed to find an optimal ordering of cities
- $O(N!)$
  - Imagine your computer can evaluate 1 million tours per second...
    - $N = 10$: ~1/3 second
    - $N = 11$: ~4 seconds
    - $N = 12$: ~40 seconds
    - $N = 13$: ~8 minutes
    - $N = 14$: ~2 hours
    - $N = 15$: 1+ day
    - ...  
    - $N = 20$: *over a million years!!*
Traveling Salesman Problem (TSP)

• Can you improve the algorithm?

• Some ideas...
  • Prune bad routes as soon as possible
    • (But what's a "bad route"?)
  • Avoid re-calculating the distance of sub-segments you've calculated before
  • Look for "good" solutions rather than optimal solutions
    • (But what's a "good" or "good enough" solution?)
Conclusions

• Algorithms are the key to computational problem solving

• The design of an algorithm can make the difference between useful and impractical

• Algorithm choice often involves tradeoffs
  • Space (memory) vs. time (speed)
  • Optimal solution vs. feasible runtime