Complexity analysis: Why should we care?

- Reasons why we shouldn't care:
  - Computing power is cheap
  - Computing power is plentiful
  - Memory is cheap
  - Memory is plentiful

Consider these examples:

- Britannica.com
- Any of these webcasts:
  - Victoria's Secret fashion show
  - Madonna concert
  - NetAid concert
  - ...

Where are the potential “slow points” in a program?

- Memory read/write/copy
- Networking (processing incoming requests from a socket, e.g.)
- Number of times a loop executes
- Number of instructions in an algorithm
- Size of the file or dataset on which we are working
Definitions

- *Computational complexity*: the “cost” of an algorithm
  - time: how much time does the algorithm take to complete?
  - space: how much memory does an algorithm use or require?
  - there are other measures, too
- We will usually consider time complexity in this class --- what is an algorithm’s *run time*?

Measuring complexity

- Instead of using real-time units, we will use *logical-time* units
  - Let $n$ be the size of the file, array, or data set on which the algorithm is operating
  - The time required for an algorithm to process this data is $f(n)$
- Q: what is $f(n)$?

How do we measure complexity?

Can we use real-time units (seconds)? Why or why not?

Objective

- We are interested in how the logical running time of an algorithm changes as the size of the data set ($n$) increases
- We are not interested in the exact running time; rather, we want to make a good approximation to the upper and lower bounds on the running time

**COMPLEXITY ANALYSIS**
Different notions of complexity

- Big-O notation (aka “order-N” notation)
- \( \Omega \) notation (“omega” notation)
- \( \Theta \) notation (“theta” notation)
- Average complexity
- Best-case complexity
- Worst-case complexity
- Amortized complexity

Order-N notation

- Provides an approximate upper bound on the running time of an algorithm
- Definition:

  We say that \( f(n) \) is order-g(n) \( [O(g(n))] \) if there exists some constant \( c > 0 \) such that

  \[
  f(n) \leq c \cdot g(n)
  \]

  for all \( n \) larger than some positive \( N \)

Example:

\[
f(n) = 5n^2 + n + 4
\]

Notes

- Our function is actually greater than the possible upper bound functions at small values of \( n \)
- We are most concerned with large (very large) \( n \)
- Thus, this function is \( O(n^2) \)
Some properties of order-N notation

- Transitive: \( f(n) = O(g(n)) \) & \( g(n) = O(h(n)) \) \(\Rightarrow\) \( f(n) = O(h(n)) \)
- Additive: \( f(n) = O(h(n)) \) & \( g(n) = O(h(n)) \) \(\Rightarrow\) \( f(n) + g(n) = O(h(n)) \)
- Power rule: \( a^n = O(n^k) \)
- Another power rule: \( n^k = O(n^{k+j}), j>0 \)
  - generally, we take the smallest power for which the equality is valid

Logarithmic functions

- Grow slowly, therefore an algorithm that is “order-logN” is very desirable
- Property: \( \log_a n = O(\log_b n) \), \( a \) and \( b \) not equal to 1
  - means that any logarithmic function is on the order of any other logarithmic function (approximately)
  - we’ll just denote this case as \( O(\lg n) \)

How fast do certain functions grow?

Examples

```java
for (count=0, i=1; i<=n; i++)
for (j=1; j<=n; j++)
count++;  // Drozdek, pg. 68-69
```

```java
for (count=0, i=1; i<=n; i++)
for (j=1; j<i; j++)
count++;  // Drozdek, pg. 68-69
```
Example: Binary search

```java
int binarySearch(int[] items, int key) {
    int low = 0;
    int mid;
    int high = items.length - 1
    while (low <= high) {
        mid = (low + high)/2;
        if (key < items[mid])
            high = mid -1; // search in lower half
        else if (items[mid] < key)
            low = mid + 1; // search in upper half
        else return mid;  // success
    }
    return -1;  // key is not in the array
}
(Drozdek, pg 59)
```

Complexity of binary search

- Best case: key is in the middle of the array
  - run time = 1 loop
- Worst case: key is not in the array
  - pare down array to size 1 by halving the array \( m \) times: \( n, n/2, n/2^2, \ldots n/2^m \)
  - \( n/2^m = 1 \rightarrow m = \log n \)
  - So the worst case complexity of binary search is \( O(\log n) \) --- which is excellent!

Average case complexity

- Usually the running time of an algorithm depends on what data we start with
  - example: searching an array
- Often we want to know the best, worst, and average case running times
- Q: How do we compute the average complexity of an algorithm?

Definition

- **Expected value** : \( E(x) = \sum p_i x_i \)
  - \( p_i \) is the probability that \( x_i \) occurs in the data
  - basically means “weighted average”
- We’ll use this to calculate average complexity
What is the average case complexity of binary search?

- Need to know
  - all possible cases
  - probability of each case occurring
- For this example, assume $n = 2^m$

Average case complexity (cont.)

- Number of tries: $1*1 + 2*2 + 3*4 + \ldots + n/2 \log n$
  $$= \sum_{i=0}^{\log n - 1} 2^i(i+1)$$
- Computing this sum is difficult!
- Strategy: find another power series that is less than or equal to this one
  - try $c \sum 2^i$
  - let $c = \log n / 2$

Average case complexity (cont.)

- Now take the average:
  - each try has probability $1/n$
  - so our sum is $(1/n) \times ((\log n)/2) \times (n - 1)$
  - $= O(\log n)!$
- So the average case complexity for binary search is the same as the worst case complexity!

Amortized complexity

- How do we find the run time of a complex algorithm?
- One method: assume worst case run time for each part of the algorithm
  - problem: may make our estimate way off
- Another method: analyze a set of operations rather than each operation in isolation
  - amortized analysis
Amortized complexity: The basic idea

- Each operation has a “cost”
- Quick (cheap) operations are “charged” more to subsidize more expensive operations
- Need to figure out what cost structure to use that overcharges the cheap operations (like insertions) enough to offset the more expensive operations (like copying to/from memory)
- We’ll come back to this idea later

Ω notation and Θ notation

- $\Omega(n)$ is a lower bound for the complexity of an algorithm
- $\Theta(n)$ is an “order of magnitude” indicator for the complexity of an algorithm
- For the most part, we’ll use upper bound and order of magnitude interchangeably when we speak of “order-N” notation