Analysis of Algorithms
To **analyze** an algorithm means:

- developing a formula for predicting *how fast* an algorithm is, based on the *size of the input* (**time complexity**), and/or
- developing a formula for predicting *how much memory* an algorithm requires, based on the *size of the input* (**space complexity**)

**Usually time** is our biggest concern

- Most algorithms require a fixed amount of space
What does “size of the input” mean?

- If we are searching an array, the “size” of the input could be the size of the array
- If we are merging two arrays, the “size” could be the sum of the two array sizes
- If we are computing the $n^{th}$ Fibonacci number, or the $n^{th}$ factorial, the “size” is $n$
- We choose the “size” to be a parameter that determines the actual time (or space) required
  - It is usually obvious what this parameter is
  - Sometimes we need two or more parameters
In computing time complexity, one good approach is to count \textit{characteristic operations}.

What a “characteristic operation” is depends on the particular problem. If searching, it might be comparing two values. If sorting an array, it might be:
- comparing two values
- swapping the contents of two array locations
- both of the above

Sometimes we just look at how many times the \textit{innermost} loop is executed.
Exact values

- It is sometimes possible, *in assembly language*, to compute *exact* time and space requirements
  - We know exactly how many bytes and how many cycles each machine instruction takes
  - For a problem with a known sequence of steps (factorial, Fibonacci), we can determine how many instructions of each type are required

- However, often the exact sequence of steps cannot be known in advance
  - The steps required to sort an array depend on the actual numbers in the array (which we do not know in advance)
In a higher-level language (such as Java), we do not know how long each operation takes.

Which is faster, $x < 10$ or $x \leq 9$?

We don’t know exactly what the compiler does with this.

The compiler almost certainly optimizes the test anyway (replacing the slower version with the faster one).

In a higher-level language we cannot do an exact analysis.

Our timing analyses will use major oversimplifications.

Nevertheless, we can get some very useful results.
Average, best, and worst cases

- Usually we would like to find the *average* time to perform an algorithm
- However,
  - Sometimes the “average” isn’t well defined
    - Example: Sorting an “average” array
      - Time typically depends on how out of order the array is
      - How out of order is the “average” unsorted array?
    - Sometimes finding the average is too difficult
- Often we have to be satisfied with finding the *worst* (longest) time required
  - Sometimes this is even what we want (say, for time-critical operations)
- The *best* (fastest) case is seldom of interest
Constant time

- **Constant time** means there is some constant $k$ such that this operation always takes $k$ nanoseconds.

- A Java statement takes constant time if:
  - It does not include a loop.
  - It does not include calling a method whose time is unknown or is not a constant.

- If a statement involves a choice (**if** or **switch**) among operations, each of which takes constant time, we consider the statement to take constant time.
  - This is consistent with **worst-case analysis**.
Linear time

- We may not be able to predict to the nanosecond how long a Java program will take, but do know *some* things about timing:

  ```java
  for (i = 0, j = 1; i < n; i++) {
    j = j * i;
  }
  ```

- This loop takes time \( k*n + c \), for some constants \( k \) and \( c \):
  - \( k \) : How long it takes to go through the loop once (the time for \( j = j * i \), plus loop overhead)
  - \( n \) : The number of times through the loop (we can use this as the “size” of the problem)
  - \( c \) : The time it takes to initialize the loop
- The total time \( k*n + c \) is *linear in* \( n \)
Constant time is (usually) better than linear time

Suppose we have two algorithms to solve a task:
- Algorithm A takes 5000 time units
- Algorithm B takes $100*n$ time units

Which is better?
- Clearly, algorithm B is better if our problem size is small, that is, if $n < 50$
- Algorithm A is better for larger problems, with $n > 50$
- So B is better on small problems that are quick anyway
- But A is better for large problems, where it matters more

We usually care most about very large problems
- But not always!
The array subset problem

- Suppose you have two sets, represented as unsorted arrays:
  ```java
  int[] sub = { 7, 1, 3, 2, 5 };
  int[] super = { 8, 4, 7, 1, 2, 3, 9 };
  ```
  and you want to test whether every element of the first set (sub) also occurs in the second set (super):
  ```java
  System.out.println(subset(sub, super));
  ```
  (The answer in this case should be false, because sub contains the integer 5, and super doesn’t)

- We are going to write method subset and compute its time complexity (how fast it is)

- Let’s start with a helper function, member, to test whether one number is in an array
```java
static boolean member(int x, int[] a) {
    int n = a.length;
    for (int i = 0; i < n; i++) {
        if (x == a[i]) return true;
    }
    return false;
}
```

- If \( x \) is \textit{not} in \( a \), the loop executes \( n \) times, where \( n = a.length \)
  - This is the \textit{worst case}
- If \( x \) \textit{is} in \( a \), the loop executes \( n/2 \) times \textit{on average}
- Either way, linear time is required: \( k*n+c \)
static boolean subset(int[] sub, int[] super) {
    int m = sub.length;
    for (int i = 0; i < m; i++)
        if (!member(sub[i], super) return false;
    return true;
}

The loop (and the call to member) will execute:
- m = sub.length times, if sub is a subset of super
  - This is the worst case, and therefore the one we are most interested in
  - Fewer than sub.length times (but we don’t know how many)
    - We would need to figure this out in order to compute average time complexity

The worst case is a linear number of times through the loop
But the loop body doesn’t take constant time, since it calls member, which takes linear time
Analysis of array subset algorithm

We’ve seen that the loop in subset executes $m = \text{sub.length}$ times (in the worst case).

Also, the loop in subset calls $\text{member}$, which executes in time linear in $n = \text{super.length}$.

Hence, the execution time of the array subset method is $m \times n$, along with assorted constants.

- We go through the loop in $\text{subset}$ $m$ times, calling $\text{member}$ each time.
- We go through the loop in $\text{member}$ $n$ times.
- If $m$ and $n$ are similar, this is roughly quadratic, i.e., $n^2$. 
What about the constants?

- An added constant, \( f(n) + c \), becomes less and less important as \( n \) gets larger.

- A constant multiplier, \( k \times f(n) \), does *not* get less important, but...
  - Improving \( k \) gives a *linear* speedup (cutting \( k \) in half cuts the time required in half).
  - Improving \( k \) is usually accomplished by careful code optimization, not by better algorithms.
  - We aren’t that concerned with *only* linear speedups!

- Bottom line: *Forget the constants!*
Simplifying the formulae

- Throwing out the constants is one of two things we do in analysis of algorithms
  - By throwing out constants, we simplify $12n^2 + 35$ to just $n^2$
- Our timing formula is a polynomial, and may have terms of various orders (constant, linear, quadratic, cubic, etc.)
  - We usually discard all but the highest-order term
    - We simplify $n^2 + 3n + 5$ to just $n^2$
Big O notation

- When we have a polynomial that describes the time requirements of an algorithm, we simplify it by:
  - Throwing out all but the highest-order term
  - Throwing out all the constants

- If an algorithm takes $12n^3 + 4n^2 + 8n + 35$ time, we simplify this formula to just $n^3$

- We say the algorithm requires $O(n^3)$ time

- We call this **Big O notation**
  - Later on we will talk about related **Big Omega** and **Big Theta**
Big O for subset algorithm

- Recall that, if $n$ is the size of the set, and $m$ is the size of the (possible) subset:
  - We go through the loop in subset $m$ times, calling member each time
  - We go through the loop in member $n$ times
- Hence, the actual running time should be $k*(m*n) + c$, for some constants $k$ and $c$
- We say that subset takes $O(m*n)$ time
Can we justify Big O notation?

- Big O notation is a *huge* simplification; can we justify it?
  - It only makes sense for *large* problem sizes
  - *For sufficiently large problem sizes, the highest-order term swamps all the rest!*

- Consider \( R = x^2 + 3x + 5 \) as \( x \) varies:

<table>
<thead>
<tr>
<th>( x )</th>
<th>( x^2 )</th>
<th>( 3x )</th>
<th>( 5 )</th>
<th>( R )</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>5</td>
<td>5</td>
</tr>
<tr>
<td>10</td>
<td>100</td>
<td>30</td>
<td>5</td>
<td>135</td>
</tr>
<tr>
<td>100</td>
<td>10000</td>
<td>300</td>
<td>5</td>
<td>10,305</td>
</tr>
<tr>
<td>1000</td>
<td>1000000</td>
<td>3000</td>
<td>5</td>
<td>1,003,005</td>
</tr>
<tr>
<td>10,000</td>
<td>10^8</td>
<td>3*10^4</td>
<td>5</td>
<td>100,030,005</td>
</tr>
<tr>
<td>100,000</td>
<td>10^10</td>
<td>3*10^5</td>
<td>5</td>
<td>10,000,300,005</td>
</tr>
</tbody>
</table>
$y = x^2 + 3x + 5$, for $x=1..10$
$y = x^2 + 3x + 5$, for $x=1..20$
Common time complexities

- \(O(1)\): constant time
- \(O(\log n)\): log time
- \(O(n)\): linear time
- \(O(n \log n)\): log linear time
- \(O(n^2)\): quadratic time
- \(O(n^3)\): cubic time
- \(O(n^k)\): polynomial time
- \(O(2^n)\): exponential time

Better vs. Worse

\[\text{BETTER} \quad \text{WORSE}\]
The End

(for now)