• Algorithm costs: time, space, and energy
• Best case, worst case, average case analysis
• Counting instructions and asymptotic analysis
• Big-O, big-Omega, big-Theta notation
• Introduction to algorithm measurement
Writing good software

• Keep in mind the characteristics of good software:
  
  – robustness: a program’s ability to spot exceptional conditions and deal with them or shutdown gracefully
  
  – correctness: does the program do what it is supposed to do?
  
  – efficiency: all programs use resources (time and space and energy, i.e. CPU cycles and memory and battery or wall power); how can we measure efficiency so that we can compare algorithms?
Efficiency and Cost Functions

An algorithm’s efficiency can be described by:

- **time** complexity or cost – how long it takes to execute. In general, less time is better!
- **space** complexity or cost – how much computer memory it uses. In general, less space is better!
- **energy** complexity or cost – how much energy uses. In general, less energy is better!

- Costs are usually given as *functions of the size of the input* to the algorithm:
  
  A big instance of the problem will probably take more resources to solve than a small one, but how much more?
Cost function measurement vs. analysis

• For a given algorithm, if the size of the input is $n$, we would like to know:
  • $T(n)$, the time cost of solving the problem
  • $S(n)$, the space cost of solving the problem
  • $E(n)$, the energy cost of solving the problem

• Two approaches:
  – We could implement the algorithm and run it and measure the time, memory, and energy usage
  – Or we can analyze the written algorithm
Algorithm cost analysis

- A way to start to do algorithm time cost analysis:
  - Write down the algorithm that solves the problem
  - Decide what “size of the problem” means for this kind of problem
  - Count up the number of instructions the algorithm would execute, as a function of $n$, the size of the problem it is solving

- But note that different languages for writing the same algorithm might require a different number of instructions
  - Java will probably require fewer than C which will require fewer than assembly language, etc.

- So, a precise count of the number of instructions is, arguably, too much detail... we will need to consider how to abstract away from nonessential detail
Algorithm cost cases

- It’s important to distinguish among different kinds of cases that can occur when running a given algorithm to solve a problem of size $n$

- **Best case**: for all inputs of size $n$, which has the lowest cost?
  - optimistic, but useful for a lower bound on cost

- **Worst case**: for all inputs of size $n$, which has the highest cost?
  - pessimistic, but useful for an upper bound on cost

- **Average case**: average the cost over all inputs of size $n$
  - useful, but can be hard to analyze
Analyzing an Algorithm

• As an example let’s consider analyzing a simple algorithm that computes the average of the values in an array

• For such an algorithm, the size of the problem is: the length of the array

• We will write down the algorithm as a Java method, and then count the number of instructions that would be executed when the method is called
## Time Cost: Counting Instructions

<table>
<thead>
<tr>
<th>Statements</th>
<th>Cost</th>
</tr>
</thead>
<tbody>
<tr>
<td>float findAvg (int []grades){{</td>
<td></td>
</tr>
<tr>
<td>float sum = 0;</td>
<td>1</td>
</tr>
<tr>
<td>int count = 0;</td>
<td>1</td>
</tr>
<tr>
<td>while (count &lt; grades.length) {</td>
<td>n+1</td>
</tr>
<tr>
<td>sum += grades[count];</td>
<td>n</td>
</tr>
<tr>
<td>count++;</td>
<td>n</td>
</tr>
<tr>
<td>}</td>
<td></td>
</tr>
<tr>
<td>if (grades.length &gt; 0)</td>
<td>1</td>
</tr>
<tr>
<td>return sum / grades.length;</td>
<td></td>
</tr>
<tr>
<td>else</td>
<td>1</td>
</tr>
<tr>
<td>return 0.0f;</td>
<td></td>
</tr>
<tr>
<td>}</td>
<td></td>
</tr>
<tr>
<td>TOTAL</td>
<td>3n+5</td>
</tr>
</tbody>
</table>

Here the size of the problem \(n\) is the length of the array.

How many times will each instruction execute, as a function of \(n\)?

Then what is the total number of instructions executed, as a function of \(n\)?
Abstract cost functions

• A time cost function that you get by precisely counting instructions is, for many purposes, really too precise
  – If you wrote the algorithm slightly differently, or in a different language, you’d get a different $T(n)$…
  – … but it’s still basically the same algorithm

• We want a more abstract characterization of cost functions

• An abstraction will ignore some details, but hopefully those details will not be too important, and the abstraction will be useful
Toward asymptotic cost analysis

• We are mainly interested in seeing how an algorithm cost function grows, as the size of the problem $n$ grows

• And we are mainly interested in characterizing this growth as $n$ becomes large, so we can compare algorithms to solve large, real-world problems
  • for small $n$, almost any algorithm will be very fast on a modern computer!

• This leads to the idea of asymptotic cost analysis: coming up with a simple cost function that qualitatively characterizes how the ‘true’ cost function grows, as a function of the problem size $n$, as $n$ becomes large
Characterizing growth rates

- Consider that time cost function we got by counting instructions: $T(n) = 3n + 5$
- A characteristic of that time cost function is that it is growing \textit{approximately linearly} as a function of $n$:
  - If you increase $n$ by a factor of 100, or 1000, the value of $T(n)$ grows by about the same factor…
  - And this is more exactly true with larger $n$

\[
\begin{align*}
T(1) &= 8 \\
T(10) &= 35 \\
T(100) &= 305 \\
T(1000) &= 30005 \\
T(10000) &= 300005 \\
T(100000) &= 3000005
\end{align*}
\]
Functions with similar growth rates

• Using this level of abstraction, functions like $3n + 5$, $1000n$, $n/2 + 77$, and $n$ are all similar to each other…
  • They grow approximately linearly (i.e., like $n$) as a function of $n$ when $n$ is large

• By the same token, functions like $3n^2 + 5$, $1000n^2$, $n^2 + 2n$, and $n^2$ are all similar to each other (and very different from those linear functions!)…
  • They grow approximately quadratically (i.e., like $n^2$) as a function of $n$ when $n$ is large

• These considerations about describing cost functions fit very well with the mathematical ideas of big-O, big-Omega, and big-Theta notation

• We will formally define these notations, and work through some examples…
Big-Oh (O) Notation defined

• We say a function $f(n)$ is “big-O” of another function $g(n)$, and write $f(n) = O(g(n))$, if:

  There are positive constants $c$ and $n_0$ such that

  $$f(n) \leq c \, g(n) \quad \text{for all} \quad n \geq n_0.$$  

• That is:
  • $f(n)$ will grow no faster than a constant times $g(n)$, for large $n$;
  • $g(n)$ provides a qualitative upper bound on the growth rate of $f(n)$;
  • For all $n \geq n_0$, the graph of $f(n)$ falls below the graph of $cg(n)$
Linear Search: 'worst case' analysis

<table>
<thead>
<tr>
<th>Statements</th>
<th>Cost</th>
</tr>
</thead>
<tbody>
<tr>
<td>int linearSearch( int []a, int target ) {</td>
<td></td>
</tr>
<tr>
<td>int i = 0; int n = a.length;</td>
<td>2</td>
</tr>
<tr>
<td>while ( i &lt; n ) {</td>
<td>n + 1</td>
</tr>
<tr>
<td>if ( target == array[i] )</td>
<td>n</td>
</tr>
<tr>
<td>return i;</td>
<td>0</td>
</tr>
<tr>
<td>i++;</td>
<td>n</td>
</tr>
<tr>
<td>}</td>
<td></td>
</tr>
<tr>
<td>return -1;</td>
<td>1</td>
</tr>
</tbody>
</table>

The maximum possible number of instructions gives the worst case, which for this algorithm happens when the target is not in the array.

TOTAL 3n + 4
Big-O example

• For the linear search algorithm, the worst case time cost is $T(n) = 3n + 4$

• Let $c = 4$, $n_0 = 4$. Then since $3n + 4 \leq 4n$ for all $n \geq 4$, it follows that $T(n) = O(n)$

• Note that by the definition of big-O, it would also be true to say $T(n) = O(100n)$, and $T(n) = O(n^2)$, etc., etc.

• … but $O(n)$ specifies a better, simpler, more informative, 'tighter' upper bound on the asymptotic growth of $T(n)$, so that it what we would usually use in this case
Big-O: Linear Search

\[ T_{\text{linearSearch}}(n) = 3n + 4 = O(n) \quad \text{in the worst case.} \]
Big-O exercises

Express the following functions of $n$ using big-O notation. In each case, try to use a simple function of $n$ to accurately characterize the asymptotic growth of the given function:

- $3 \log_2 n + 4 n \log_2 n + n$
- $546 + 34n + 2n^2$
- $2^n + 14n^2 + 4n^3$
- 100
Big-omega notation defined

• We say a function $f(n)$ is “big-omega” of another function $g(n)$, and write $f(n) = \Omega(g(n))$, if:
  There are positive constants $c$ and $n_0$ such that $f(n) \geq c g(n)$ for all $n \geq n_0$.

• That is:
  • $f(n)$ will grow no slower than a constant times $g(n)$;
  • $g(n)$ provides a qualitative lower bound on the growth rate of $f(n)$;
  • For all $n \geq n_0$, the graph of $cg(n)$ falls below the graph of $f(n)$.
Big-theta notation defined

• We say a function $f(n)$ is “big-theta” of another function $g(n)$, and write $f(n) = \Theta(g(n))$, if:

\[ f(n) = O(g(n)) \text{ and } f(n) = \Omega(g(n)) \]

• Therefore $g(n)$ is qualitatively a tight, upper and lower bound on the growth rate of $f(n)$. 
Big theta notation: an equivalent def'n

- A function \( f(n) \) is \( \Theta(g(n)) \) if there are positive constants \( c_1, c_2, \) and \( n_0 \) such that
  \[
  0 \leq c_1 \ g(n) \leq f(n) \leq c_2 \ g(n)
  \]
  for all \( n \geq n_0 \).

- This means that for all \( n \geq n_0 \), the graph of \( f(n) \) falls between \( c_1 \ g(n) \) and \( c_2 \ g(n) \).
Theta (Θ) Example: findAvg()

- \( T_{\text{findAvg}}(n) = \Theta(n) \): for \( c_1 = 2 \), \( c_2 = 4 \), \( n_0 = 5 \)

For all \( n \geq 5 \),
\[
2n \leq 3n + 5 \leq 4n
\]
int medianOf3( int []a, int n ) {
    int v1 = a[0];
    int v2 = a[n/2];
    int v3 = a[n-1];
    if ( (v1 < v2) && (v1 < v3) ) // v1 is smallest
        if (v2 < v3)
            return n/2; // middle position
        else return n-1; // last position
    else if ( (v2 < v1) && (v2 < v3) ) // v2 smallest
        if (v1 < v3)
            return 0; // first position
        else return n-1; // last position
    else // v3 is smallest
        if (v1 < v2)
            return 0; // first position
        else return n/2; // middle position
}

O(1): the cost is independent of the size of the problem

Question: What is the best case big-O time cost of linear search?
Cost function classes

- When using big-O, big-Omega, big-Theta notation to state cost functions, the goal is to write down a simple function that usefully characterizes the growth of the cost function.
  - “Simple” means the function you write usually has just one term, with no constant coefficient.
- Some common functions of \( n \) that are used for this purpose are shown in the following table.
- The table shows how these functions of \( n \) grow compared to \( n \) itself.
- Their growth rates are quite different!
### Some Common Cost Function Classes

<table>
<thead>
<tr>
<th>( \log_2 n )</th>
<th>( n )</th>
<th>( n \log_2 n )</th>
<th>( n^2 )</th>
<th>( 2^n )</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2</td>
<td>2</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>2</td>
<td>4</td>
<td>8</td>
<td>16</td>
<td>16</td>
</tr>
<tr>
<td>3</td>
<td>8</td>
<td>24</td>
<td>64</td>
<td>256</td>
</tr>
<tr>
<td>4</td>
<td>16</td>
<td>64</td>
<td>256</td>
<td>65,536</td>
</tr>
<tr>
<td>5</td>
<td>32</td>
<td>160</td>
<td>1,024</td>
<td>4,294,967,296</td>
</tr>
<tr>
<td>6</td>
<td>64</td>
<td>384</td>
<td>4,096</td>
<td>( 1.84 \times 10^{19} )</td>
</tr>
<tr>
<td>7</td>
<td>128</td>
<td>896</td>
<td>16,384</td>
<td>( 3.40 \times 10^{38} )</td>
</tr>
<tr>
<td>8</td>
<td>256</td>
<td>2,048</td>
<td>65,536</td>
<td>( 1.16 \times 10^{77} )</td>
</tr>
<tr>
<td>9</td>
<td>512</td>
<td>4,608</td>
<td>262,144</td>
<td>( 1.34 \times 10^{154} )</td>
</tr>
<tr>
<td>10</td>
<td>1,024</td>
<td>10,240</td>
<td>1,048,576</td>
<td>( 1.80 \times 10^{308} )</td>
</tr>
</tbody>
</table>
Algorithm analysis vs. measurement

- Asymptotic analysis (just counting statements executed, and stating the result as a simple function using big-O, big-Omega, or big-Theta notation) is elegant, and it's important to know how to do it... but it doesn't tell the full story.

- For example: in terms of asymptotic analysis of time cost, finding the largest value in an array of int's and an array of Integer objects is the same, but in reality...

- So sometimes you should consider algorithm measurement, also known as benchmarking
Algorithm Measurement

- To perform algorithm measurement:
  - Implement the algorithm
  - Decide what “size of the problem” means for this kind of problem
  - Create instances of the problem of different sizes, run the algorithm on these instances, and measure the time it takes

- The resulting data should give you a good idea of the actual time cost function of the algorithm in practice

- But it can be tricky to get good measurements...
Basic Algorithm Measurement

Pseudocode: for timing a data structure algorithm

1. for problem size $N = \text{min,...max}$
2. initialize the data structure
3. get the starting time
4. run the algorithm on problem size $N$
5. get the finish time
6. $\text{elapsed time} = \text{finish time} - \text{start time}$
7. output elapsed time on problem size $N$
Some Interesting Results

<table>
<thead>
<tr>
<th>$n$</th>
<th>array of int</th>
<th>array of Integer</th>
</tr>
</thead>
<tbody>
<tr>
<td>800,000</td>
<td>2.314</td>
<td>4.329</td>
</tr>
<tr>
<td>4,000,000</td>
<td>11.363</td>
<td>21.739</td>
</tr>
<tr>
<td>8,000,000</td>
<td>22.727</td>
<td>42.958</td>
</tr>
</tbody>
</table>

- From these measurements, what is the apparent big-O time cost of `findMax()` on array of int?_______ on array of Integer?_______
- Why would array of Integer take more time?
Next time

- Implementations of the List ADT
- Properties of array and linked implementations
- Separate and inner classes
- Singly and doubly linked lists
- Evaluating and selecting a data structure

Reading: Gray, Ch 3