Lecture 6

Complexity

- Lab 1 — Preview
- Time-Complexity
- Big-Oh
- Addendum: Arrays vs Linked Lists
Preview of Lab 1
The Coin Strip Lab involves designing and implementing a data structure for the Silver Dollar Game.
Code for playing a 1-Player Puzzle or a 2-Player Game is given to you (e.g. `Game.java` on left). But it won’t run until you design and implement the `CoinStrip` data structure (start code on right).
Time-Complexity
Live Coding: Time Complexity

We’ll discuss the basics of time counting during the live coding of the Count.java file.

- What if we change the static values CAPACITY and/or MAXVALUE? What if we make CAPACITY ten times larger? How much longer will the program run?
- Does second implementation of the loop save a significant amount of time? What exactly would we mean by significant?
- If // do something was simple (e.g., print(i)), then how much would the if statement contribute to the overall run-time? What if it was more complicated?
- How much time and space does new int[CAPACITY] take?
import java.util.Random;

public class Count {
    public static void main(String[] args) {
        final int CAPACITY = 100;
        final int MAXVALUE = 1000;

        int[] A = new int[CAPACITY];
        Random rand = new Random();

        // How much time does this loop take?
        for (int i = 0; i < CAPACITY; i++) {
            A[i] = rand.nextInt(MAXVALUE);
        }

        // How much time does this loop take?
        // How much faster is this loop?

        // How much time does this if-statement take?
        // How much time did the whole program take?
        // How much space did the whole program take?
    }
}
import java.util.Random;

public class Count {
    public static void main(String[] args) {
        final int CAPACITY = 100;
        final int MAXVALUE = 1000;

        int[] A = new int[CAPACITY];
        Random rand = new Random();

        // How much time does this loop take?
        for (int i = 0; i < CAPACITY; i++) {
            A[i] = rand.nextInt(MAXVALUE);
        }

        // How much time does this loop take?
        for (int i = 0; i < CAPACITY; i++) {
            for (int j = 0; j < CAPACITY; j++) {
                if (i != j && A[i] == A[j]) {
                    System.out.printf("A[%d] = A[%d] = %d\n", i, j, A[i]);
                }
            }
        }
        System.out.println("\n");

        // How much faster is this loop?
        for (int i = 0; i < CAPACITY-1; i++) {
            for (int j = i+1; j < CAPACITY; j++) {
                if (i != j && A[i] == A[j]) {
                    System.out.printf("A[%d] = A[%d] = %d\n", i, j, A[i]);
                }
            }
        }
        System.out.println("\n");
    }
}

The loop runs \textit{CAPACITY} times. Each iteration takes the same time.

The outer loop runs \textit{CAPACITY} times. On each iteration, the inner loop runs \textit{CAPACITY} times. So the \textit{if} statement runs \textit{CAPACITY} * \textit{CAPACITY} times.

Intuitively, this runs about half as long as the previous one since it only generates \textit{i} and \textit{j} pairs with \textit{i} < \textit{j}.

Alternatively, let's add the time for each iteration of the \textit{i} loop.

The first iteration (i.e., \textit{i} = 0) runs \textit{CAPACITY} - 1 times. The second (i.e., \textit{i} = 1) runs \textit{CAPACITY} - 2 times, etc.

Recall that $1 + 2 + \ldots + n = \frac{n(n+1)}{2}$.

So the \textit{if} statement runs $(\text{\textit{CAPACITY}} - 1) \times \text{\textit{CAPACITY}} / 2$ times in total.

\textbf{Finished Count.java.}
This will be true 50% of the time.

In this branch, the loop runs \texttt{CAPACITY} times.

In this branch, the loop runs \texttt{MAXVALUE} times.

A pessimistic view would take the maximum of the two branches: \(\max(\texttt{CAPACITY}, \texttt{MAXVALUE})\).
Run-Time versus Time-Complexity

Run-Time
The time that a program takes when it is run.
Measured in a time unit (e.g. milliseconds, days).
- Consider the unix command `time`

We can estimate the run-time using basic rules.
- How long does each instruction take? Often we estimate each to be one unit of time.
- Sequential instructions are added together.
- A loop is estimated by multiplying the number of loops by the time taken for each loop.
- Take the maximum of the two branches in an `if`-statement.

The amount of time is often *parameterized* based on certain values.

Time-Complexity
A more abstract measurement of the time complexity.

The focus is on how much time is taken relative to the size of the problem, or more specifically, the size of the input denoted \( n \).
- Larger problems take longer to solve.

We use big-oh to provide more useful and concise measurements, and the analysis may involve simple proofs.

Time-complexity is often pessimistic (i.e., it considers worst-case performance).

Uses the same principles as run-time counting.
Big-Oh
From the CSCI 136 textbook *Java Structures*. 
We are interested in how the run-time of an algorithm $f(n)$ grows as $n \to \infty$. 

---

**Figure 5.2** Near-origin details of common curves. Compare with Figure 5.3.

**Figure 5.3** Long-range trends of common curves. Compare with Figure 5.2.
From the CSCI 136 textbook *Java Structures*. We bound the run-time $f(n)$ using a simpler function $g(n)$.

Figure 5.1 Examples of functions, $f(n)$, that are $O(g(n))$. 
**Big-Oh**

Exact runtime formulae are overly complicated. We instead use big-oh notation as an estimate. Big-oh is an upper bound that does two things:

- Remove lower order (i.e., slower growing) terms.
- Remove constant factors.

When measuring time we count each individual simple step as 1.

- Use addition for consecutive operations.
- Use multiplication for loops.

It’s more accurate to refer to $O(f(n))$ as a set, e.g. $10n^2 + 3n \in O(n^2)$ or $10n^2 + 3n \in O(n^3)$.

---

**Definition 5.1** A function $f(n)$ is $O(g(n))$ (read “order g” or “big-O of g”), if and only if there exist two positive constants, $c$ and $n_0$, such that

$$|f(n)| \leq c \cdot g(n)$$

for all $n \geq n_0$.

**DEFINITION 7.2**

Let $f$ and $g$ be functions $f, g : \mathbb{N} \rightarrow \mathbb{R}^+$. Say that $f(n) = O(g(n))$ if positive integers $c$ and $n_0$ exist such that for every integer $n \geq n_0$,

$$f(n) \leq c \cdot g(n).$$

**Asymptotic Upper Bounds** Let $T(n)$ be a function—say, the worst-case running time of a certain algorithm on an input of size $n$. (We will assume that all the functions we talk about here take nonnegative values.) Given another function $f(n)$, we say that $T(n)$ is $O(f(n))$ (read as “$T(n)$ is order $f(n)$”) if, for sufficiently large $n$, the function $T(n)$ is bounded above by a constant multiple of $f(n)$. We will also sometimes write this as $T(n) = O(f(n))$. More precisely, $T(n)$ is $O(f(n))$ if there exist constants $c > 0$ and $n_0 \geq 0$ so that for all $n \geq n_0$, we have $T(n) \leq c \cdot f(n)$. In this case, we will say that $T$ is asymptotically upper-bounded by $f$. It is important to note that this definition requires a constant $c$ to exist that works for all $n$; in particular, $c$ cannot depend on $n$.

Formal definitions of big-oh from various sources.
Growth of various functions (scale: n = 0 – 100).

- Notice that $n^2$ and $2^n$ don’t look too different on this scale.
Growth of various functions (scale: \( n = 0 - 1000 \)).

- Notice that \( n^2 \) and \( 2^n \) are starting to look very different on this scale.
Variants of Big-Oh

We also use the following variants of Big-Oh.

<table>
<thead>
<tr>
<th>Bound</th>
<th>Notation</th>
<th>Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>≤</td>
<td>$O$</td>
<td>big oh</td>
</tr>
<tr>
<td>&lt;</td>
<td>$o$</td>
<td>little oh</td>
</tr>
<tr>
<td>≥</td>
<td>$\Omega$</td>
<td>big omega</td>
</tr>
<tr>
<td>&gt;</td>
<td>$\omega$</td>
<td>little omega</td>
</tr>
<tr>
<td>=</td>
<td>$\Theta$</td>
<td>big theta</td>
</tr>
</tbody>
</table>

For example, we’ll argue that comparison-based sorting takes $\Omega(n \log(n))$-time later in the course.
Examples
Polynomials

Polynomials have the form \( c_0 + c_1 n + c_2 n^2 + \ldots + c_k n^k \)
where the \( c \)'s are constant and \( k \) is a constant.

Note: A constant function is a (special type of) polynomial function.

Polynomials have excellent closure properties.
If \( p(x) \) and \( q(x) \) are both polynomials:

- \( c \cdot p(x) \) is a polynomial for any constant \( c \).
- \( p(x) + q(x) \) is a polynomial.
- \( p(x) \cdot q(x) \) is a polynomial.
- \( p(q(x)) \) is a polynomial (known as composition).

The last point implies that you can call a polynomial-time subroutine a polynomial number of times, and the resulting run-time will still be polynomial.

Note: Terms of the form \( c_k n^k \log(n) \) are polylogarithmic with \( n \log(n) \) arising frequently (e.g., merge sort).

<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
</tr>
</thead>
<tbody>
<tr>
<td>11</td>
<td>12</td>
<td>13</td>
<td>14</td>
<td>15</td>
<td>16</td>
<td>17</td>
<td>18</td>
<td>19</td>
<td>20</td>
<td></td>
</tr>
<tr>
<td>21</td>
<td>22</td>
<td>23</td>
<td>24</td>
<td>25</td>
<td>26</td>
<td>27</td>
<td>28</td>
<td>29</td>
<td>30</td>
<td></td>
</tr>
<tr>
<td>31</td>
<td>32</td>
<td>33</td>
<td>34</td>
<td>35</td>
<td>36</td>
<td>37</td>
<td>38</td>
<td>39</td>
<td>40</td>
<td></td>
</tr>
<tr>
<td>41</td>
<td>42</td>
<td>43</td>
<td>44</td>
<td>45</td>
<td>46</td>
<td>47</td>
<td>48</td>
<td>49</td>
<td>50</td>
<td></td>
</tr>
<tr>
<td>51</td>
<td>52</td>
<td>53</td>
<td>54</td>
<td>55</td>
<td>56</td>
<td>57</td>
<td>58</td>
<td>59</td>
<td>60</td>
<td></td>
</tr>
<tr>
<td>61</td>
<td>62</td>
<td>63</td>
<td>64</td>
<td>65</td>
<td>66</td>
<td>67</td>
<td>68</td>
<td>69</td>
<td>70</td>
<td></td>
</tr>
<tr>
<td>71</td>
<td>72</td>
<td>73</td>
<td>74</td>
<td>75</td>
<td>76</td>
<td>77</td>
<td>78</td>
<td>79</td>
<td>80</td>
<td></td>
</tr>
<tr>
<td>81</td>
<td>82</td>
<td>83</td>
<td>84</td>
<td>85</td>
<td>86</td>
<td>87</td>
<td>88</td>
<td>89</td>
<td>90</td>
<td></td>
</tr>
<tr>
<td>91</td>
<td>92</td>
<td>93</td>
<td>94</td>
<td>95</td>
<td>96</td>
<td>97</td>
<td>98</td>
<td>99</td>
<td>100</td>
<td></td>
</tr>
</tbody>
</table>

How many ordered pairs of distinct values?
How many unordered pairs of distinct values?

Answers: \( n(n-1) \) and \( n(n-1)/2 \).
Exponential Functions

Exponential terms have the form \( c \cdot b^{e(n)} \) where \( e(n) \) is some polynomial in \( n \). When comparing these to polynomials, the variable \( n \) appears in the exponent rather than in the base.

Notes:
- The base of the exponent is important. For example, \( 3^n \) grows much faster than \( 2^n \) or \( 2^{10n} \).
- If \( e(n) \) is logarithmic, then \( c \cdot b^{e(n)} \) is polynomial.
- If \( e(n) \) is exponential, then \( c \cdot b^{e(n)} \) is doubly exponential.

How many subsets of numbers are there?  
How many permutations of numbers?  
Answers: \( 2^n \) and \( n! \)
Logarithms commonly arise in computer science when we repeatedly halve the search space. (In some cases the halving is obtained using the help of a data structure.)

Notes:
- \( \log \log(x) \) is not the same as \( \log^2(x) \).
- By default the base is assumed to be 2.
- We typically don’t care about the base of the logarithm. This is due to the change of base formula which results in a constant factor.

\[
\log_a n = \frac{\log_b n}{\log_b a}
\]
Constant Functions

There are relatively few problems that can be answered in constant time. One example is below.

**FIRST**

<table>
<thead>
<tr>
<th>Input:</th>
<th>A non-empty list of integers L, and an integer k.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Output:</td>
<td>yes, if the first integer in L is k. Otherwise, no.</td>
</tr>
</tbody>
</table>

Constant functions also arise when analyzing problems whose input/instance size is bounded.

For example, the number of humans is $n \leq 8,000,000,000$, which is constant, so sorting human names can be done in constant-time. In particular, $n \log n = 32,000,000,000$.

This limitation is inherent to the way that we analyze algorithms. Claiming constant-time for every real-world instance is a great way to annoy computer scientists!
Addendum: Arrays vs Linked Lists
Sequential Data

There are two fundamental ways in which sequential data is stored in a computer.

1. **Arrays.** The values are positioned sequentially within memory.
   - Pros: Fast access based on position.
   - Cons: Homogeneous (data points have same size); fixed predetermined size; slow insert / delete.

2. **Linked Lists.** The values are chained together using pointers.
   - Pros: Easy to resize; fast insert / delete.
   - Cons: Slow access based on position; node types.

Linked lists have variations including (a) doubly linked, (b) circularly linked, (c) tail pointers. We’ll look at these in more detail later in the course.