# CSCI-1200 Data Structures — Fall 2025
# Lecture 8 — Algorithm Analysis a.k.a. Big O Notation

## Review from Lecture 7

- Designing our own container classes
- Dynamically-allocated memory in classes
- Copy constructors, assignment operators, and destructors *We ran out of time, we'll finish this today!*
- Templated classes, Implementation of the DS `Vec` class, mimicking the STL `vector` class

## Today's Lecture

- Algorithm Analysis / Computational Complexity
- Orders of Growth, Formal Definition of Big O Notation

## 8.1 Algorithm Analysis / Computational Complexity

*Why should we bother?*

- We want to do better than just implementing and testing every idea we have.
- We want to know why one algorithm is better than another.
- We want to know the best we can do. (This is often quite hard.)

*How do we do it?* There are several options, including:

- Don't do any analysis; just use the first algorithm you can think of that works.
- Implement and time algorithms to choose the best.
- Analyze algorithms by counting operations while assigning different weights to different types of operations based on how long each takes.
- Analyze algorithms by assuming each operation requires the same amount of time. Count the total number of operations, and then multiply this count by the average cost of an operation.

## 8.2 Exercise: Counting Example

- Suppose `arr` is an array of `n` doubles. Here is a simple fragment of code to sum of the values in the array:

```
double sum = 0;
for (int i=0; i<n; ++i)
  sum += arr[i];
```

- What is the total number of operations performed in executing this fragment? Come up with a function describing the number of operations *in terms of n*.

## 8.3 Exercise: Which Algorithm is Best?

A venture capitalist is trying to decide which of 3 startup companies to invest in and has asked for your help. Here's the timing data for their prototype software on some different size test cases:

| n | foo-a | foo-b | foo-c |
|---|---|---|---|
| 10 | 10 u-sec | 5 u-sec | 1 u-sec |
| 20 | 13 u-sec | 10 u-sec | 8 u-sec |
| 30 | 15 u-sec | 15 u-sec | 27 u-sec |
| 100 | 20 u-sec | 50 u-sec | 1000 u-sec |
| 1000 | ? | ? | ? |

Which company has the "best" algorithm?

## 8.4 Formal Definition: Big O Notation -Growth Rate, Order of the Function

*In this course we will focus on the intuition of big O notation. This topic will be covered again, in more depth, in later computer science courses.*

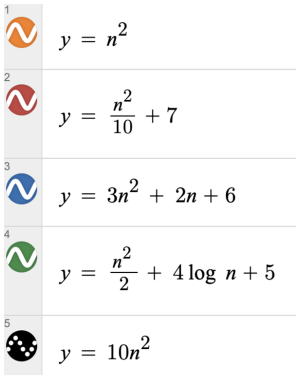
- Definition: Algorithm $A$ is order $f(n)$ — denoted $O(f(n))$ — if constants $k$ and $n_0$ exist such that $A$ requires no more than $k * f(n)$ time units (operations) to solve a problem of size $n \geq n_0$.
- For example, algorithms requiring $3n + 2$, $5n - 3$, and $14 + 17n$ operations are all $O(n)$.

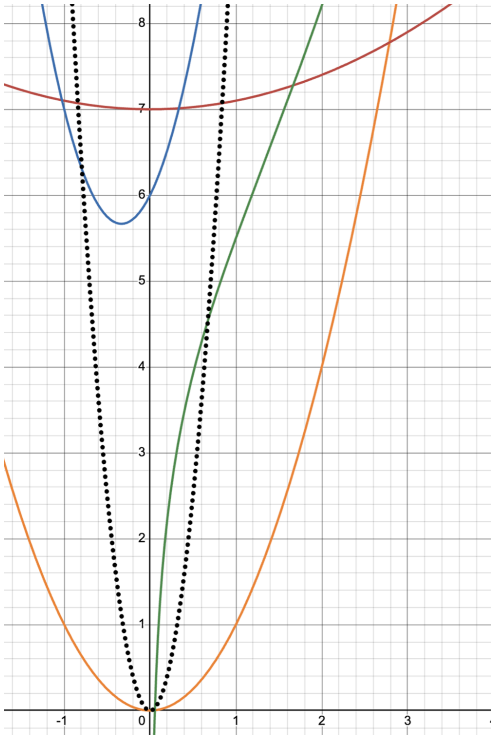
  This is because we can select values for $k$ and $n_0$ such that the definition above holds. (What values?)

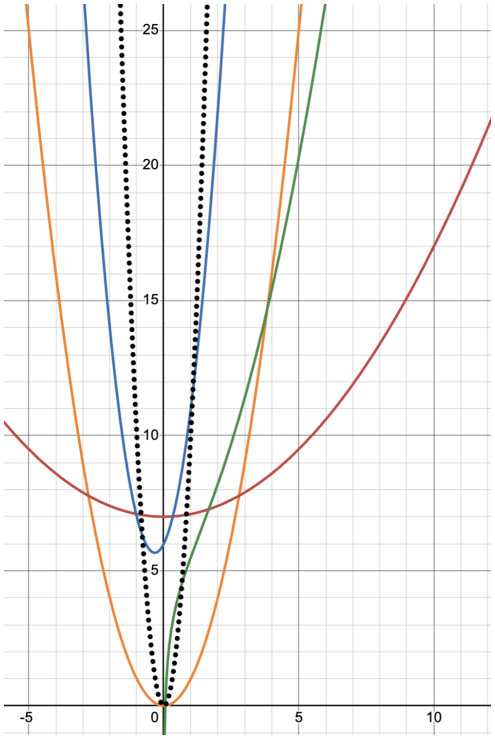
  Likewise, algorithms requiring $n^2/10 + 15n - 3$ and $10000 + 35n^2$ are all $O(n^2)$.
- Intuitively, we determine the order by finding the *asymptotically dominant term (function of n)* and throwing out the leading constant. This term could involve logarithmic or exponential functions of $n$. Implications for analysis:
  - We don't need to quibble about small differences in the numbers of operations.
  - We also do not need to worry about the different costs of different types of operations.
  - We don't produce an actual time. We just obtain a rough count of the number of operations. This count is used for comparison purposes.
- In practice, this makes analysis relatively simple, quick and (sometimes unfortunately) rough.

## 8.5 Simplifying a Big O Notation Expression – An Example using Pictures!

- Crossing out the non-dominant terms and the constant coefficient in front of the dominant term, the red, blue, and green functions (left image below) are all simplified to be part of the same Big O Notation = $O(n^2)$.
- Plotting these functions together (middle image below) it is not clear how they are related to or bounded by the orange function $y = n^2$.




- For this specific group of functions, it is sufficient to set $k$ to 10, and zoom out a little bit (right image above). We can see that the dotted black line bounds the other functions from above for $n > n_0 = 2$.
- And for any other specific function that belongs in $O(n^2)$, we can find a $k$ and $n_0$ such that $k * n^2$ bounds that function from above for values $n > n_0$. Just increase $k$ and/or $n_0$ and zoom out as needed.

## 8.6 Common Orders of Magnitude

- $O(1)$, *a.k.a. CONSTANT*: The number of operations is independent of the size of the problem. e.g., compute quadratic root.
- $O(\log n)$, *a.k.a. LOGARITHMIC.* e.g., dictionary lookup, binary search.
- $O(n)$, *a.k.a. LINEAR.* e.g., sum up a list.
- $O(n \log n)$, e.g., sorting.
- $O(n^2)$, $O(n^3)$, $O(n^k)$, *a.k.a. POLYNOMIAL.* e.g., find closest pair of points.
- $O(2^n)$, $O(k^n)$, *a.k.a. EXPONENTIAL.* e.g., Fibonacci, playing chess.

## 8.7 Exercise: A Slightly Harder Example

- Here's an algorithm to determine if the value stored in variable `x` is also in an array called `foo`. Can you analyze it? What did you do about the `if` statement? What did you assume about where the value stored in `x` occurs in the array (if at all)?

```
int loc=0;
bool found = false;
while (!found && loc < n) {
  if (x == foo[loc]) found = true;
  else loc++;
}
if (found) cout << "It is there!\n";
```

## 8.8 Best-Case, Average-Case and Worst-Case Analysis

- For a given fixed size array, we might want to know:
  - The fewest number of operations (best case) that might occur.
  - The average number of operations (average case) that will occur.
  - The maximum number of operations (worst case) that can occur.
- The last is the most common. The first is rarely used.
- On the previous algorithm, the best case is $O(1)$, but the average case and worst case are both $O(n)$.

## 8.9 Approaching An Analysis Problem

- Decide the important variable (or variables) that determine the "size" of the problem. For arrays and other "container classes" this will generally be the number of values stored.
- Decide what to count. Big O notation helps us here.
  - If each loop iteration does a fixed (or bounded) amount of work, then we only need to count the number of loop iterations.
  - We might also count specific operations. For example, in the previous exercise, we could count the number of comparisons.
- Do the count and use big O notation to describe the result.

## 8.10 Exercise: Big O Notation

For each code fragment below, give the big O notation estimate of the number of operations as a function of `n`:

1.
```
int count=0;
for (int i=0; i<n; ++i)
  for (int j=0; j<n; ++j)
    ++count;
```

2.
```
int count=0;
for (int i=0; i<n; ++i)
  ++count;
for (int j=0; j<n; ++j)
  ++count;
```

3.
```
int count=0;
for (int i=0; i<n; ++i)
  for (int j=i; j<n; ++j)
    ++count;
```

## 8.11 What is the Big O Notation for `vector::push_back`?

- What variable(s) should we assign to the input data to describe the size of this problem?

```
template <class T> void Vec<T>::push_back(const T& val) {
  if (m_size == m_alloc) {
    m_alloc *= 2;
    if (m_alloc < 1) m_alloc = 1;
    T* new_data = new T[ m_alloc ];
    for (size_type i=0; i<m_size; ++i)
      new_data[i] = m_data[i];
    delete [] m_data;
    m_data = new_data;
  }
  // Add the value at the last location and increment the bound
  m_data[m_size] = val;
  ++ m_size;
}
```

- What is the worst case running time?
  Analyze the different pieces (line-by-line, loops, function calls),
  Combine them (using addition or multiplication as appropriate), and then
  Simplify that expression.

- But what if we make 1,000 or 1,000,000 calls to `push_back` on the same vector object?
  Some calls are expensive, but some are cheap. What is the *average cost*?

## 8.12 Problem from Last Night's Exam: Letter Search at the Zoo

- The Problem: "Write a function named `LetterSearch` that accepts 3 arguments named `words`, `letter`, and `selected`. The function should examine the strings in the `words` vector, collecting all strings that contain the character `letter` in the `selected` vector."

- Let's analyze the **Instructor Solution** to this problem:

```
void LetterSearch(const std::vector<std::string> &words, char letter,
                  std::vector<std::string> &selected) {
  // clear out the vector of any previous answer
  selected.clear();
  for (unsigned int i = 0; i < words.size(); i++) {
    bool flag = false;
    for (int j = 0; j < words[i].size(); j++) {
      if (words[i][j] == letter) {
        flag = true;
        break;
      }
    }
    if (flag)
      selected.push_back(words[i]);
  }
}
```

- What variable(s) should we assign to the input data to describe the size of this problem?

- Count the operations – look for loops or expensive helper functions.
  Determine how to combine the pieces – do they add or multiply?
  Write down the expression, then simplify.
  What is your final answer?

- What if we instead used the `find` function, which was NOT allowed on the exam.

```
void LetterSearch(const std::vector<std::string> &words, char letter,
                  std::vector<std::string> &selected) {
  // clear out the vector of any previous answer
  selected.clear();
  for (unsigned int i = 0; i < words.size(); i++) {
    if (words[i].find(std::string(1,letter)) != std::string::npos) {
      selected.push_back(words[i]);
    }
  }
}
```

- Would our Big O Notation answer be different?

## 8.13 So about last night's test...

- The graduate TAs, some of the mentors, and myself are grading the exam this afternoon/evening. *Today's TA normal Office Hours 4-6pm are canceled.* The graded tests will be released sometime tomorrow (Saturday). Sample solutions will be posted on the website.

- **Talk to the instructor:**
  My office hours are Mondays 1-3pm in Lally 302 and after lecture Tuesday & Friday. Please stick around after lecture. I do get to everyone. We can have a semi-private conversation.

- **How did you feel during the test?**
  Were you rushed? Did you run out of time? Did you understand the problem? Did you know how to solve the problem, but you didn't know how to make it fit in the provided boxes? Did you know how to solve the problem, but only in a language other than C++?

- **Were you surprised by your grade on the test?**
  After looking at the solutions, do you understand why your answers were marked the way they were marked?

- **What did you to do prepare for the test?**
  Did you do the practice problems? Did you fully write out your answers on paper? Did you time yourself? (A good estimate is # of points = # of minutes.)

- **My recommendations for how to prepare for future tests:**
  After writing out your answer on paper, you can also type them up and compile & test & debug. The problems are short and shouldn't take much additional code to run. This is a good exercise to catch and learn from small syntax mistakes. Also, it will help you understand bugs in logic, loop bounds, etc.

- **"I want an 'A'. Give me an extra assignment so I can get an 'A'." or "I need a 'B'. What do I need to do to get a 'B'?" or "How can I do the least work and guarantee I get a 'C-'?"**
  These are the wrong questions. They make all instructors grumpy. Better questions are:"I don't seem to understand const & reference, can you explain them again?" Or "I'm lost on Big O Notation, I understand the simple examples but I don't understand how to approach using it on a general problem."

- **If you would like to switch to Computer Science 1**
  Email `cs1instructors@cs.lists.rpi.edu` ASAP. Even though it is after add/drop date, you can petition for a late registration change. CS1's first test is next week Thursday night 6-8pm. They will likely not accept new students after the first test.