Keenan Knaur
Adjunct Lecturer

California State University, Los Angeles
Computer Science Department

Algorithm Analysis and Big O Notation

"algorithm (n.): a word used by programmers when they do not want to explain what they did."

CS2013: Programming with Data Structures

What is an Algorithm?

- ***algorithm***: an ordered sequence of steps that is unambiguous, executable, and terminating.
  - ***unambiguous***: the sequence of steps must be clear with no guessing as to which step comes next.
  - ***executable*: the sequence of steps must be able to be implemented and executed by a computer.**
  - ***terminating****: the sequence of steps must must eventually read an end.*

- An algorithm is **NOT** a program / software.
  - This is a common misconception.
  - programs are able to loop forever, an algorithm must have an end.

- An algorithm is a recipe that describes how to perform some action.

How do we write algorithms?

- algorithms can be written using different techniques:
  - pseudocode
  - flow diagrams and other charts
  - ordered list of steps

- Example Algorithm: Making an omelette:
  - 1) Get the frying pan.
  - 2) Get the oil.
    - a. Do we have oil?
      - i. If yes, put oil in the pan.
      - ii. If no, do we want to buy oil?
        - 1. If yes, go buy oil.
        - 2. If no, terminate making omlette.
  - 3) Turn on the stove, etc...

How to Compare Algorithms?

- Comparison based on **execution time**:
  - Implement the algorithm, and insert lines of code to see the running time.
  - Example in Java:

```
long startTime = System.currentTimeMillis();
//algorithm code
long endTime = System.currentTimeMillis();
long elapsed = endTime – startTime;
```

  - You would then run this code with various inputs of different sizes and then plot the results.

Execution Time Analysis: Two String Methods Example

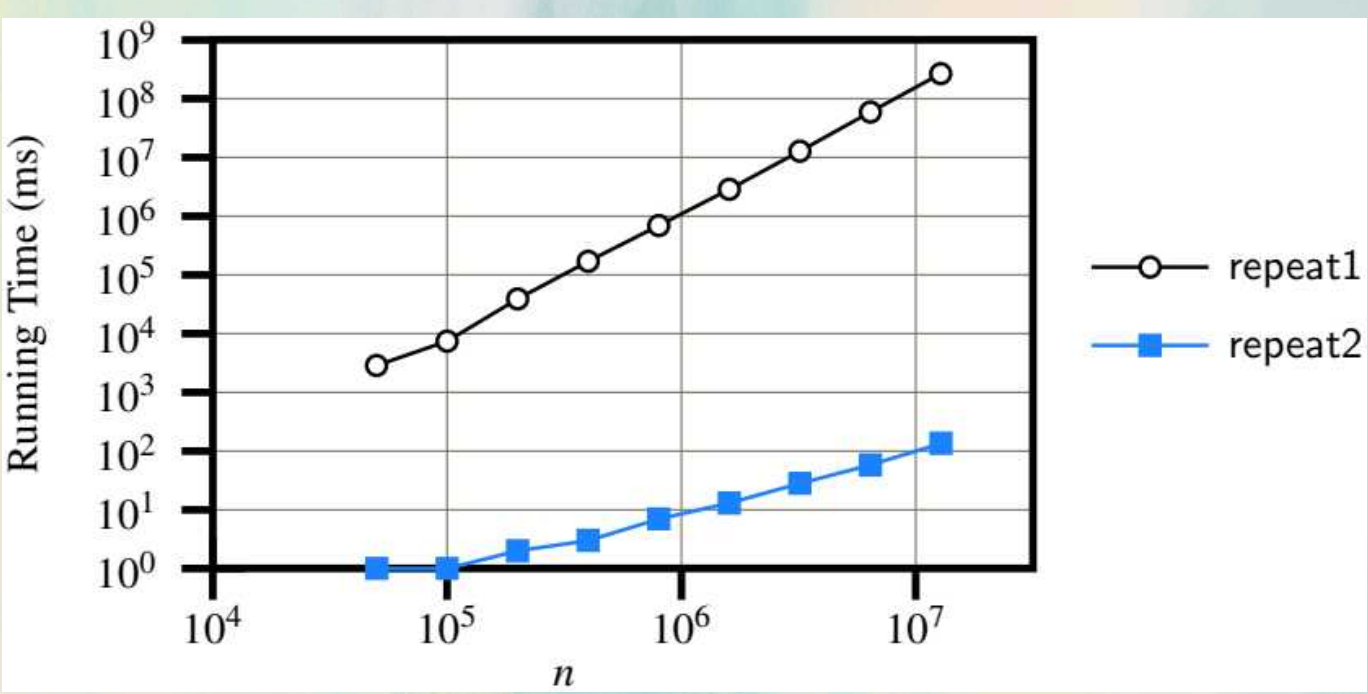
- The textbook presents a very nice example of execution time analysis by comparing two algorithms for building strings.
- One algorithm uses the concatenation operator, the other uses operation from the **`StringBuilder`** class.
- Each algorithm repeatedly builds a string composed of character c.

```
1  /** Uses repeated concatenation to compose a String with n copies of character c. */
2  public static String repeat1(char c, int n) {
3    String answer = "";
4    for (int j=0; j < n; j++)
5      answer += c;
6    return answer;
7  }
8
9  /** Uses StringBuilder to compose a String with n copies of character c. */
10 public static String repeat2(char c, int n) {
11   StringBuilder sb = new StringBuilder();
12   for (int j=0; j < n; j++)
13     sb.append(c);
14   return sb.toString();
15 }
```

Code Fragment 4.2: Two algorithms for composing a string of repeated characters.

Execution Time Analysis: Two String Methods Example

- Results:



n	repeat1 (in ms)	repeat2 (in ms)
50,000	2,884	1
100,000	7,437	1
200,000	39,158	2
400,000	170,173	3
800,000	690,836	7
1,600,000	2,874,968	13
3,200,000	12,809,631	28
6,400,000	59,594,275	58
12,800,000	265,696,421	135

Execution Time Analysis: Challenges

- Execution Time Analysis can be a valuable tool when fine-tuning production-quality code, but there are some major limitations to this analytical approach:
  - Execution times are difficult to directly compare unless the experiment is performed using the exact same hardware and software environments.
    - Results will vary greatly from machine to machine and even from experiment to experiment depending on what software is running in the background.
  - You can only use a limited set of test inputs, you may leave out running times of inputs not included, and these inputs may be important.
  - You have to fully implemented the algorithm to study its running time.

How to Compare Algorithms?

- Comparison based on **number of statements (primitive operations) executed**.
  - A primitive operation could be any of the following:
    - assigning a value to a variable
    - creating an object
    - performing an arithmetic operation
    - comparisons
    - etc...
  - Count all of these operations and this can give you some idea of the time it would take to execute the total algorithm.

Number of Statements Analysis: Challenges

- While not as informative as execution time analysis, this can still be useful for a quick idea of how your algorithm may perform.

- Limitations:
  - The number of statements can vary between different programming languages.
    - Java code can usually be written using fewer lines of code in Python for example.
  - The number of statements can vary from programmer to programmer.
    - Every programmer has their own coding style, and each programmer may split their code into fewer or more statements depending on their style.
  - It could be possible to have an algorithm with many statements, but which runs more efficiently than an algorithm with fewer statements.

- In this example, **`find1()`** has far fewer statements, than **`find2()`**.

- Which algorithm is more efficient?

```
public static boolean find1(int[] arr, int key) {
    for (int i = 0 ; i < arr.length ; i++) {
        if (key == arr[i]) {
            return true;
        }
    }

    return false;
}

public static boolean find2(int[] arr, int key) {
    int low = 0;
    int high = arr.length - 1;

    while(low <= high) {
        int mid = (high + low) / 2;

        if(key == arr[mid]) {
            return true;
        }
        else if (key < arr[mid]) {
            high = mid - 1;
        }
        else if (key > arr[mid]) {
            low = mid + 1;
        }
    }
    return false;
}
```

Measuring Operations as a Function of Input Size

- To truly capture how an algorithm changes based on the size of its input, we can associate to each algorithm a mathematical function, ***f(n)*** which characterizes the number of primitive operations performed as a function of the input size ***n***.

- The idea of expressing an algorithm as a mathematical function gets us closer to the idea of Big O notation.

- Usually we focus on the ***worst-case*** running time since this is easier to identify and can lead to better algorithms.
  - If an algorithm can be written to perform well in even the most worst of cases, implies that the algorithm will do well on every type of input.

- This is usually the best and most common way to analyze algorithms....

Big O(n) Notation Made Easy!

Introduction

- Big O time analysis is a metric that computer scientists use to describe the efficiency of algorithms.

- It expresses an algorithm in terms of a function f(n) and how that algorithm / function changes as n increases to greater and greater sizes.

An Analogy of Time Complexity

- *Cracking the Coding Interview*, presents a very good analogy to explain time complexity.

- Let's say you have a file on a hard drive that you want to send to your friend who lives on the other side of the country and they need this file ASAP.

- How do you send the file?
  - email?, FTP?, some other kind of electronic transfer?

- If the file is small:
  - Obviously you could choose the fastest electronic transfer method.

- What if the file is very large...say 1 terabyte:
  - Would it be faster just to physically deliver the hard drive? A flight could be 5 - 10 hours long.
  - In some cases, yes, physically delivering the hard drive would be faster since a 1TB transfer could potentially take more than a day to electronically send.
  - What if there were no flights and you had to drive? Still this could be a faster option.

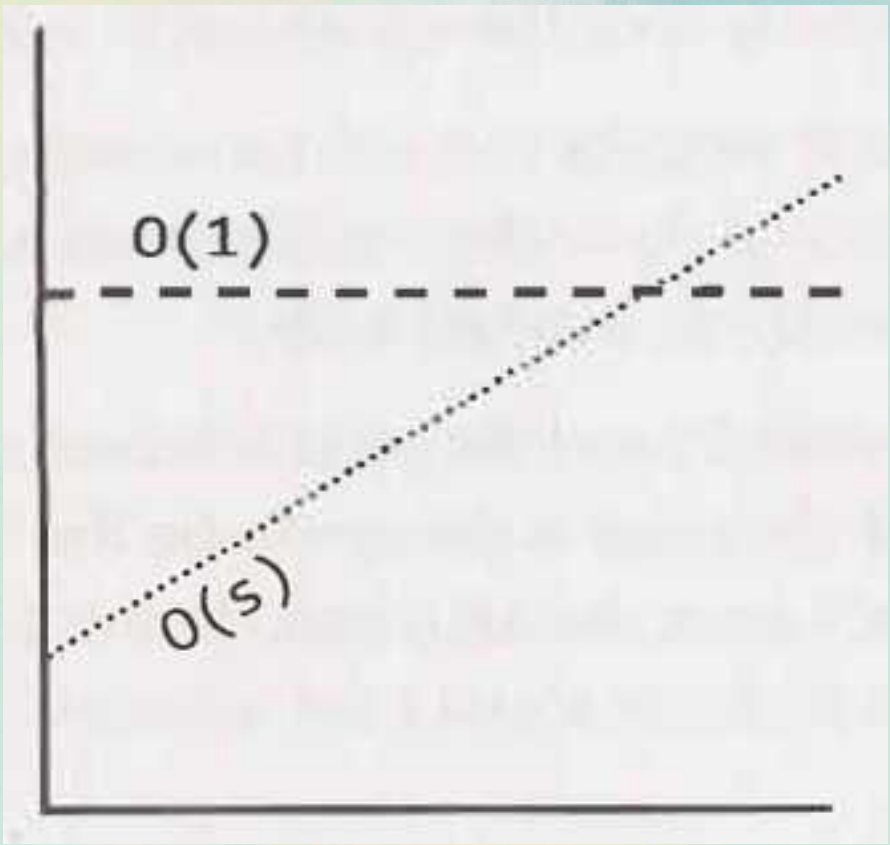
Time Complexity

- The previous analogy is what we mean by time complexity also known as asymptotic runtime or Big O time.

- The data transfer algorithm could be described as:
  - Electronic Transfer: O(s) where s is the size of the file. The would mean that the time to transfer the file increases linearly with the size of the file. (As the file size increases, so to does the electronic transfer time.)
  - Airplane Transfer: O(1) with respect to the size of the file. As the size of the file increases, it does not take us any longer to take a plane to deliver the file. The time is constant and is unaffected by the file size.

Time Complexity

- Regardless of the size of the constant or how slow the linear increase is, eventually there will be a point where the linear surpasses the constant.



Rate of Growth

- ***rate of growth***: the rate at which the running time increases as a function of the input.
  - i.e. As you have larger and larger input data sets, the running time increases at a predictable rate which we can reduce to a function.

Algorithms as Functions (no not the functions in code)

- Assume that for any given algorithm, we can condense that algorithm down to some mathematical function f(n).
  - Example, lets say we have an algorithm that can be described by the function:
    f(n) = n^2 + 78n + logn + 100.

- Assume that there also exists a more general function, g(n) which we know the behavior/characteristics of and which is easier to analyze. Also assume that no matter how complex our f(n) is, we can say that f(n) behaves characteristically like g(n).
  - Example, lets say that ***g(n) = n^2*** and we know from basic algebra this is a quadratic function and we know how to plot it very easily.

Algorithms as Functions

- ***f(n) = n^2 + 78n + logn + 100***.

input	n^2	78n	logn	100
10	100	780	3.321928	100
100	10000	7800	6.643856	100
1000	1000000	78000	9.965784	100
10000	100000000	780000	13.287712	100

Algorithms as Functions (no not the functions in code)

- The Big O notation of an algorithm says that the algorithm is **of the order of** g(n) even though the f(n) function may be very complex.
  - ***f(n) = n^2 + 78n + logn + 100***
  - ***g(n) = n^2***
  - There will exist some constant c such that cg(n) >= f(n). This means we can find a constant that will guarantee that f(n) will never increase to be greater than cg(n).
  - If the above can be proven (and usually it can be proven very easily), then we can conclude that our algorithm described by f(n) is ***of the order of*** *g(n),* that the f(n) function behaves like a g(n) function.
    - We would say that f(n) is O(n) (***of the order of n***).

Tight Bounds

- The **tight bound** means that the g(n) function is as close to our f(n) function as it can get without going below the function (if it is a tight upper bound) or going above the function (if it is a tight lower bound)

- Example: An algorithm which searches for a value in an array could be characterized as O(n), but it could also be described as $O(n^2)$, $O(n^3)$ or even other Big O times.

- We try to use the tightest bound possible to describe our algorithm, because higher bounds usually don't give us any helpful information.
  - Example: Let's say John is x years old and we can assume that no person lives past the age of 130. Then you can say that x <= 130. However, it would also be correct to say that x <= 1000 or x <= 100,000, but neither of these is really more useful or accurate for relating John's age to some upper bound.

Big O, Big Theta, and Big Omega

- All of these notations are used to describe runtimes.
- O (Big O): In academia, Big O represents the **tight upper bound** on the runtime.
- Ω (Big Omega): represents the **tight lower bound** on the runtime.
- Θ (Big Theta): represents both O and Ω and usually means equal to the function f(n).
- NOTE: In the industry people seem to have merged the definitions of Θ and O together so for our purposes in this lecture, we will use the industry definition where Big O means the tightest bound possible.

Best, Worst, and Expected Cases

- Let's use Quick Sort to describe these cases.
  - Quick Sort picks a random element which is the "pivot" and then swaps values such that elements less than the pivot appear before elements greater than the pivot. The algorithm then recursively continues.

- ***best case***: If all elements are equal, Quick Sort just goes through the array once.

- ***worst case***: That if the pivot is always chosen as the largest element?, the recursion does not divide the array in half, it just shrinks the sub array by one element. This gives us a bad runtime.

- ***average (expected) case***: Most of the time we won't have a best or worst case scenario, we will have the average or expected case. This is the case we usually assume and analyze.

Space Complexity

- Not only do we analyze runtime, but we also consider how much memory, or space, is required by the algorithm.

- Examples:
  - creating an array of size n will require O(n) space.
  - creating a 2D array of size n x n will require $O(n^2)$ space.

- Stack space counts too. Anytime a new method is called it is added to the stack. If this is a recursive method it will take up space until the currently running level of recursion is over.

The Big Seven Functions

The Big Seven

- There are seven important functions which we use in the analysis of algorithms:
  - The Constant Function
  - The Logarithm Function
  - The Linear Function
  - The NLogN Function
  - The Quadratic Function
  - The Cubic Function and Other Polynomials
  - The Exponential Function

- The previous functions are ranked from fastest to slowest runtime.

The Constant Function

- This is the simplest function of the seven:
  - ***f(n) = c***
  - c is a fixed constant such a c = 1, c = 42, c = 10^{100}
- For any input to the function *n*, the result of the function will always be a constant.
  - It doesn't matter what *n* is, the result will always be the same and does not depend on n.

The Constant Function

- If our algorithm has a constant runtime, we say that it is **O(1)**.

- Regardless of the size of the data set, the algorithm will always take a constant time
  - Example: 1 item takes 1 second, 10 items take 1 second, 100 items takes 1 second
  - always the same amount of time.

- A single statement usually runs in constant time.

```
int x = a + b + 1;
```

The Logarithm Function

- The ***logarithm function*** is defined as:
  - $f(n) = \log_b n$ for some constant b > 1.

- This function is the inverse of a power
  - $x = \log_b n$ iff $b^x = n$

- ***b*** is the base of the logarithm and we usually use a base of 2 since the computer stores things in binary.
  - lgn = $\log_2 n$
  - but sometimes in algorithm analysis we may just write logn meaning the base is understood to be 2.

The Logarithm Function

- If an algorithm runs in logarithmic time we say it is **O(logn)**

- When an algorithm takes a constant time to cut the problem size by a fraction (usually by half), this is usually logarithmic.

- The time needed increases with the data set, but not in proportion to the data set.
  - Example: 1 item takes 1 second, 10 items takes 2 seconds, 100 items takes 3 seconds.

```
while (n > 1) {
  n = n / 2;
}
```

The Linear Function

- The ***linear function*** is defined as:
  - ***f(n) = n***

- Given an input value n, the linear function assigns the value of n as the result.

- If an algorithm is linear, we say it is **O(n)**.

- The larger the dataset, the time taken grows proportionately.
  - Example: 1 item takes 1 second, 10 items takes 10 seconds, 100 items takes 100 seconds.

- Single (non-nested) loops run in Linear Time (assuming the loop is not dividing the problem into smaller and smaller subproblems.)

```
for(int i = 1 ; i <= n ; i++) {
    sum += i;
}
```

The N-Log-N or Linear Logarithmic Function

- The n-log-n function is defined as:
  - ***f(n) = nlogn***
  - For an input n, the functions result is n times the log base 2 of n.

- This function grows a little more rapidly than the linear function and a lot less rapidly than the quadratic function.

- These types of algorithms are usually algorithms that combine an O(n) operation with an O(logn) operation.

- If an algorithm is linear logarithmic time we say it is **O(nlogn).**

The N-Log-N or Linear Logarithmic Function

- Some sorting algorithms are classified as O(nlogn):
  - Usually there are 2 parts to the sort, the first loop runs in O(n) and the second loop is O(logn) which combine to form O(nlogn).
  - Example: 1 item takes 2 seconds, 10 items takes 12 seconds, 100 items takes 103 seconds.

- Mergesort and Quicksort are two sorting algorithms that are O(nlogn).

The Quadratic Function

- The ***quadratic function*** is defined as:
  - ***f(n) = n^2***
  - Given an input value of n, the result is n squared.

- This function most commonly relates to algorithms that use nested loops where the inner loop performs a linear number of operations and the outer loop is performed a linear number of times.
  - So n * n = n^2

```
for(int i = 1 ; i <= n ; i++) {
   for (int j = 1 ; j <= n ; j++) {
      x += 1;
   }
}
```

- Algorithms that have a quadratic runtime are **O(n^2)**.

- Here we start slowing down quite a bit as the running time increases faster as we add more data.
  - Example: 1 item takes 1 second, 10 items takes 100 seconds, 100 items takes 10000 seconds.

The Cubic Function and other Polynomials

- The ***cubic function*** is defined as:
  - **$f(n) = n^3$**

- The easiest example of cubic runtime in programming is when you have triple nested loops each running n times.

```
for(int i = 1 ; i <= n ; i++) {
   for (int j = 1 ; j <= n ; j++) {
      for (int k = 1; k <= n ; k++) {
         x += 1;
      }
   }
}
```

- Algorithms that have a cubic runtime are **$O(n^3)$**.

The Cubic Function and other Polynomials

- There can also be higher order polynomials in algorithm analysis, $O(n^4)$, $O(n^5)$, and so on.

- These are rarer because they are very inefficient. All you need to remember is that as the power of the polynomial increases, the algorithms becomes more and more inefficient.

The Exponential Function

- The ***exponential function*** is defined as:
  - $f(n) = b^n$
  - $f(n) = 2^n$ is the most comment one we use.

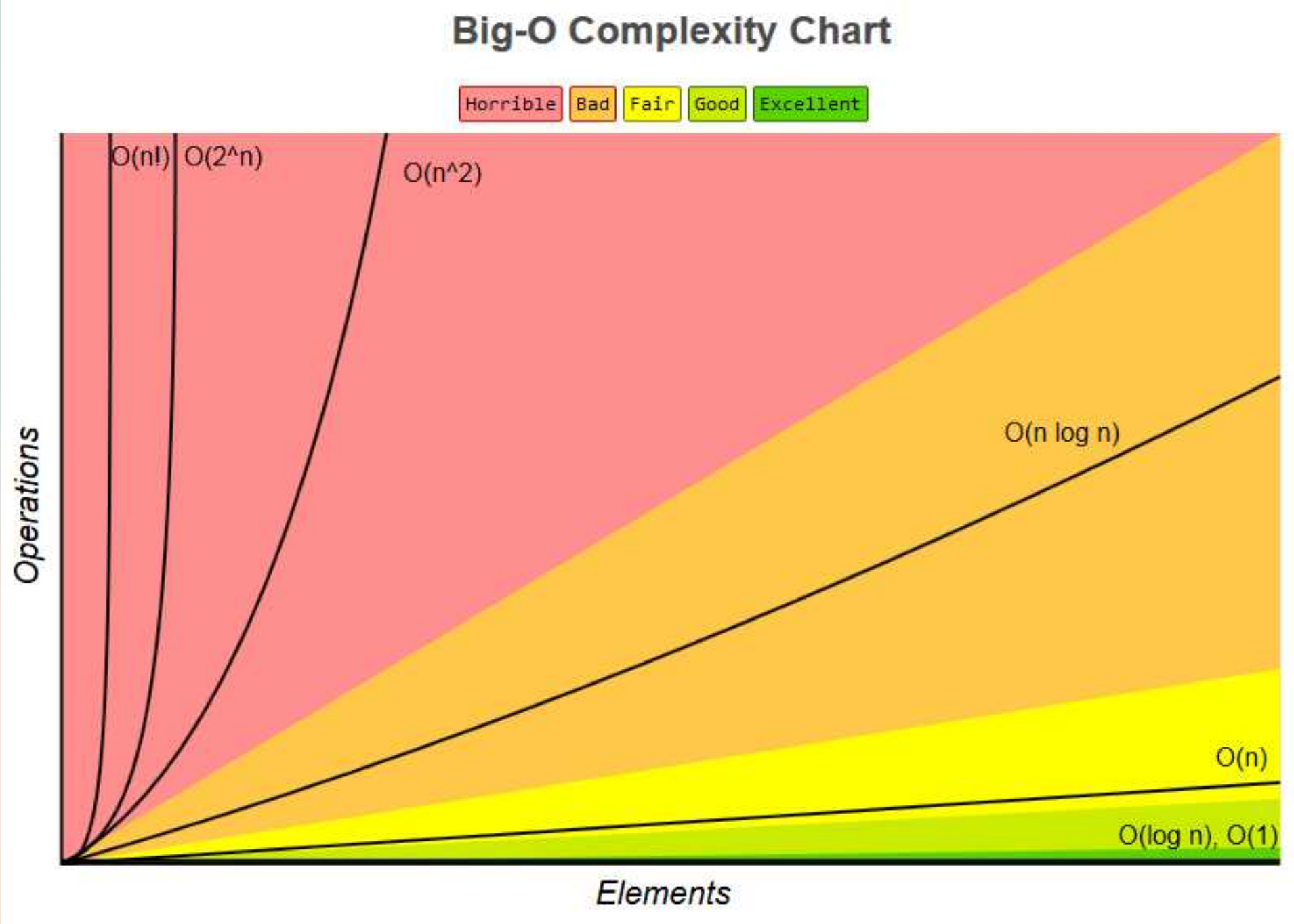
- An algorithm that has an exponential runtime is $O(2^n)$.

- This is one of the worst running times. It doesn't take very long to vastly increase the time the algorithm takes to finish by slightly increasing the size of the data set.

- Generally the algorithm takes twice as long for each new element added.
  - Example: 1 item takes 1 second, 10 items takes 1024 seconds, 100 items takes 1267650600228229401496703205376 seconds.

- Trying to crack a password by generating all possible combinations (brute force algorithm).

Growth Rate Comparison



More Analysis Hints and Examples

Drop the Constants

- Remember, it is entirely possible that code which runs in O(n) time can run faster than O(1) code. Big O just describes the ***rate of increase*** not necessarily how fast (in terms of time) an algorithm really runs.

- Because of this, we usually drop any constants so an algorithm that might be described as O(2n) is actually O(n)

Drop the Non-Dominant Terms

- What about something like: $O(n^2 + n)$?

- In a polynomial, the highest degree term is the term that has the highest exponent, and we know that the highest degree term dominates over the others when compared.

- Because the highest term dominates and means that or algorithm will always have a rate of growth equal to the highest term, we usually drop any non-dominant terms from the analysis.

- Examples:
  - $O(n^2 + n) = O(n^2)$
  - $O(n + \log n) = O(n)$
  - $O(5 * 2^n + 1000n^{100}) = O(2^N)$.

Exceptions

- You might still have a sum in a runtime where you cannot drop any terms.

- $O(B^2 + A)$ cannot be reduced (without some special knowledge of A and B).

Add vs. Multiply

- A common confusion in algorithm analysis is when do I add the runtimes vs when do I multiply the runtimes.

- Consider:

Add the Runtimes: O(A + B)

```
1   for (int a : arrA) {
2     print(a);
3   }
4
5   for (int b : arrB) {
6     print(b);
7   }
```

Multiply the Runtimes: O(A*B)

```
1   for (int a : arrA) {
2     for (int b : arrB) {
3       print(a + ",” + b);
4     }
5   }
```

- In the first example, the first loop does some work, and then the second loop does some work, so the total amount of work is O(A + B).

- In the second example the inner loop does B chunks of work for each element in A. The total amount of work is O(A * B)

Amortized Time

- Amortized analysis is used in algorithm analysis.

- The idea with an amortized analysis is instead of focusing on worst-case performance, we look at the average performance.

- ***average performance:***
  - Under certain conditions an algorithm will perform at its best
  - Under other conditions, it will perform at its worst
  - On the average an algorithm will perform somewhere in between. We call this the ***amortized time***.

Amortized Time: HashTable Example

- As we will see this semester, a Hash Table is a very efficient data structure because items can inserted into the table in O(1) constant time. (This is very good!)

- Hash Tables have to deal with collisions, which is when two items can hash to the same hash value. Since the hash value helps determine where the item is stored, we cannot store two items at the same location. One way to deal with collisions is to keep trying locations in the Table until an empty location is found. If you have to go through the entire table to find an empty location, the runtime of insertion is O(n) linear time.

- We could say that the maximum ***possible*** runtime of insertion is O(n), but this only happens in the worst case and the worst case doesn't happen very often.

- If we amortize the time and ignore the worst case, we can say the average runtime or amortized runtime is O(1)

- Conclusion:
  - Hash Table insertion CAN be O(n).
  - Hash Table amortized insertion is O(1).

Example 1

- What is the runtime?:

```
void foo(int[] array) {
   int sum = 0;
   int product = 1;

   for (int i = 0; i < array.length; i++) {
      sum += array[i];
   }

   for (int i = 0; i < array.length; i++) {
      product *= array[i];
   }

   System.out.println(sum + ", " + product);
}
```

Example 2

- What is the runtime:?

```
void printPairs(int[] array) {
   for (int i= 0; i < array.length; i++) {
      for (int j = 0; j < array.length; j++) {
         System.out.println(array[i] + "," + array[j]);
      }
   }
}
```

Example 3

- What is the runtime?:

```
void print(int[] array) {
   for (int i= 0; i < array.length; i++) {
      for (int j = i + 1; j < array.length; j++) {
         System.out.println(array[i] + "," + array[j]);
      }
   }
}
```

Example 4

- What is the runtime?:

```
void print(int[] arrayA, int[] arrayB) {
   for (int i = 0; i < arrayA.length; i++) {
      for (int j = 0; j < arrayB.length; j++) {
         if (arrayA[i] < arrayB[j]) {
            println(arrayA[i] + "," + arrayB[j]);
         }
      }
   }
}
```

Example 5

- What is the runtime?:

```
void print(int[] arrayA, int[] arrayB) {
   for (int i = 0; i < arrayA.length; i++) {
      for (int j = 0; j < arrayB.length; j++) {
         for (int k = 0; k < 100000; k++) {
            println(arrayA[i] + "," + arrayB[j]);
         }
      }
   }
}
```

Example 6

- What is the runtime?:

```
void reverse(int[] array) {
   for (int i = 0; i < array.length / 2; i++) {
      int other= array.length - i - 1;
      int temp= array[i];
      array[i] = array[other];
      array[other] = temp;
   }
}
```

Example 7

- Which of the following are equivalent to O(n)? Why?
  - O(n + p), where p < (n / 2)
  - O(2n)
  - O(n + logn)
  - O(n + m)

Example 8

- What is the runtime?:

```
int f(int n) {
   if (n <= 1) {
      return 1;
   }
   return f(n - 1) + f(n - 1);
}
```

Example 9

- What is the runtime?:

```
int sum(Node node) {
   if (node == null) {
      return 0;
   }
   return sum(node.left) + node.value + sum(node.right);
}
```

Example 10

- What is the runtime?:

```
boolean isPrime(int n) {
   for (int x = 2; x * x <= n; x++) {
      if (n % X == 0) {
         return false;
      }
   }
   return true;
}
```

Example 11

- What is the runtime?:

```
int factorial(int n) {
   if (n < 0) {
      return -1;
   }
   else if (n == 0) {
      return 1;
   }
   else {
      return n * factorial(n - 1)
   }
}
```

Example 12

- What is the runtime?:

```
void permutation(String str) {
   permutation(str, "");
}

void permutation(String str, String prefix) {
   if (str.length() == 0) {
      System.out.println(prefix);
   }
   else {
      for (int i= 0; i < str.length(); i++) {
         String rem = str.substring(0, i) +
                      str.substring(i + 1);
         permutation(rem, prefix + str.charAt(i));
      }
   }
}
```

Example 13

- What is the runtime?:

```
int fib(int n) {
   if (n <= 0) return 0;
   else if (n == 1) return 1;
   return fib(n - 1) + fib(n - 2);
}
```

Example 14

- What is the runtime?:

```
void allFib(int n) {
   for (int i = 0; i < n; i++) {
      System.out.println(i + ": "+ fib(i));
   }
}

int fib(int n) {
   if (n <= 0) return 0;
   else if (n == 1) return 1;
   return fib(n - 1) + fib(n - 2);
}
```

Example 15

- What is the runtime?:

```
void allFib(int n) {
   int[] memo = new int[n + 1];

   for (int i= 0; i < n; i++) {
      System.out.println(i + ": "+ fib(i, memo));
   }
}

int fib(int n, int[] memo) {
   if (n <= 0) return 0;
   else if (n== 1) return 1;
   else if (memo[n] > 0) return memo[n];

   memo[n]= fib(n - 1, memo) + fib(n - 2, memo);
   return memo[n];
}
```

Example 16

- What is the runtime?:

```
int powers0f2(int n) {
   if (n < 1) {
      return 0;
   }
   else if (n == 1) {
      System.out.println(l);
      return 1;
   } else {
      int prev = powers0f2(n / 2);
      int curr = prev * 2;
      System.out.println(curr);
      return curr;
   }
}
```

References

- Goodrich: Chapter 04

- Laakmann: Cracking the Coding Interview, Chapter 06