MREN 178
Data Structures and Algorithms

Week 4

Algorithm Analysis

Matthew Pan, PhD
Department of Electrical and Computer Engineering
matthew.pan@queensu.ca

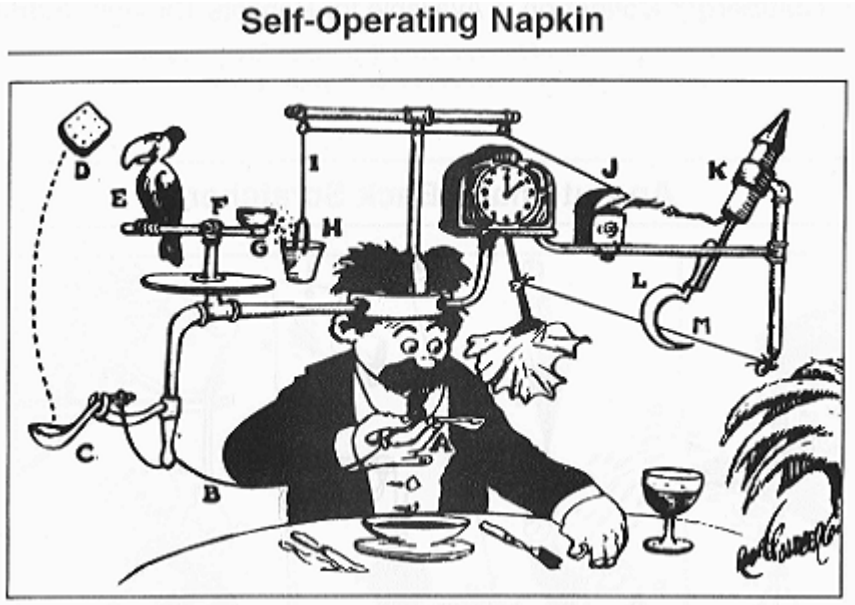
MREN 178
Data Structures and Algorithms

Analysis of Algorithms [Ch 6]

Motivation

- You may have come to realize that there are multiple ways of solving a problem
  - Some are better than others

- Example: Summation
  - Add the numbers 1 through 100
    - 1+2+3+4….+100



Example: Summation (1)

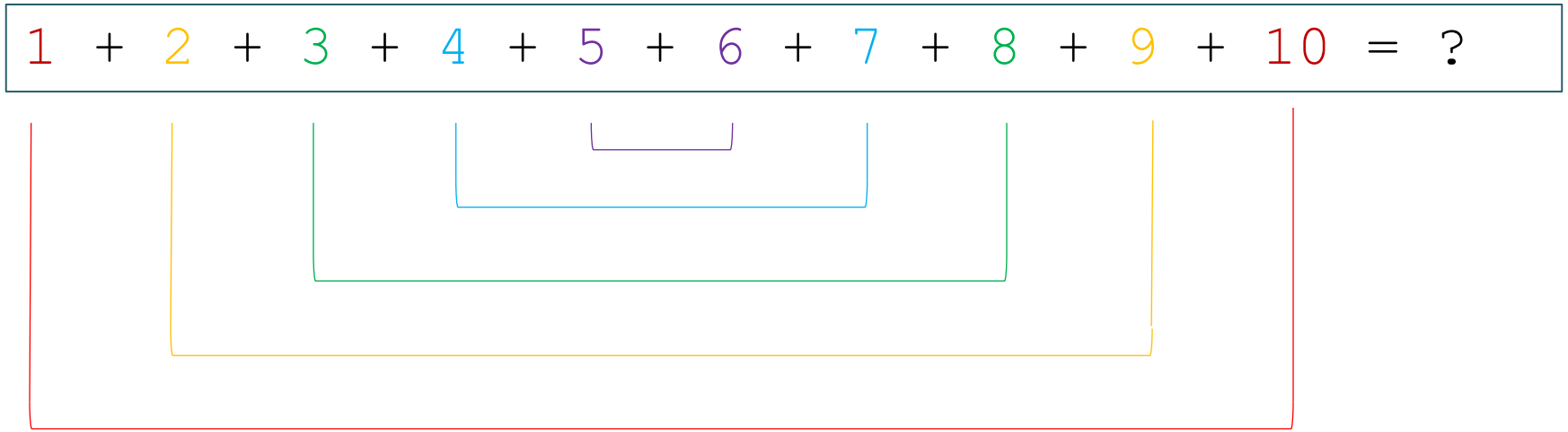
- Let’s make the problem a bit simpler for demonstration's sake:
  - Add 1 through 10
- Method 1: Brute Force

```
1 + 2 + 3 + 4 + 5 + 6 + 7 + 8 + 9 + 10 = 55
```

  - How much time will this take?

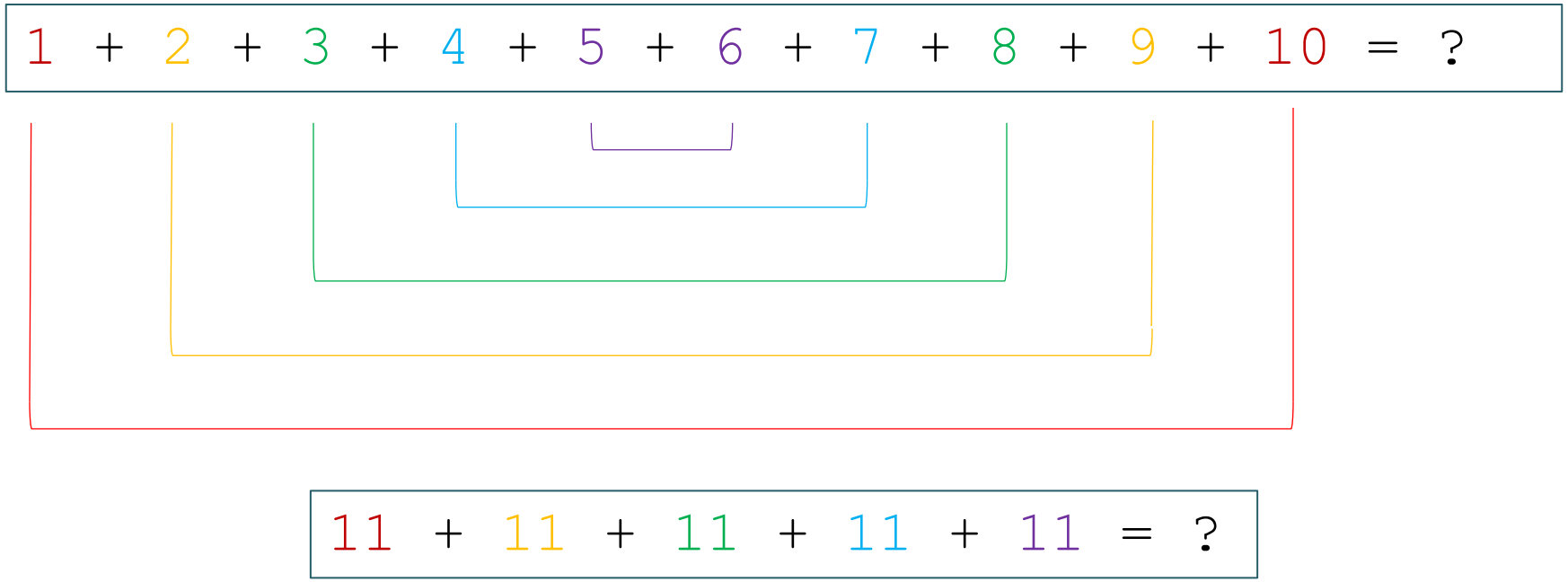
Example: Summation (2)

- Method 2: Gauss Summation



Example: Summation (3)

- Method 2: Gauss Summation



What is 1 through 100 summed?

Example: Summation (3)

- Method 2: Gauss Summation
  - Solving arithmetic sequence

```
total_sum = (no_pairs) x (sum_pairs)
```

- Summary: we compared two methods to solve a problem – one that takes significantly more effort and time, and one that takes less effort and time.
  - We want to be able to make similar comparisons between algorithms as well.

Carl Friedrich Gauss

Algorithm Comparisons and Metrics (1)

- We need ways of comparing software/code based on what's important.
  - Meeting Specifications
    - E.g., size of executable
  - Correctness
    - Is method/implementation correct and produces correct result?
    - Generally easy to test for.
  - Maintenance
    - Internal documentation (i.e., comments), modularization, readability
  - Speed
    - People get bored waiting for some code to process data.
    - People get injured because the computer system in the self-driving car cannot process image data fast enough to make a decision which would prevent an accident.
  - Resources Utilized
    - There is a finite amount of memory, processing cores, power, etc. For this course, we focus on space/memory.

Algorithm Comparisons and Metrics (2)

- First three criteria is a qualitative assessment that is subject to organizational culture (style guides, code reviews, quality assurance)
- Last two – space efficiency and time efficiency – can be analyzed quantitively

- Space complexity – (memory) space needed to run to completion
- Time complexity– time needed to run to completion
  - Time to complete payroll calculation – measure overall time using minutes and hours, perhaps.
  - Fuel injection calculation – based on accelerator position, temperature, humidity, latency of mechanical injector, etc. – need to calculate in 5 ms or less.

Space Complexity (1)

- Space needed by a program
  - Base – space needed that is NOT dependent on inputs and outputs, e.g., hello-world program.
  - Variable – space needed that changes depending on inputs and outputs, e.g., Gaussian summation where we input start of and end of integer series
- Space complexity is a concern when space is limited, e.g., on an Arduino.
  - Sometimes space complexity is traded off with time complexity, e.g., paging systems used to 'fake' very large memory – in exchange for taking a bit more time.

Space Complexity (2) - Examples

- Program gets value N, computes factorial N
  - N stack frames required
- Program gets N numbers and creates an ordered collection of those numbers
  - N or 2N depending on method
- Program reads data representing points on an N x M grid and searches grid for patterns.
  - Space will be NxM for worst case, assume N==M, so space required is N^2

Space Complexity (3)

- Space complexity is generally less of a concern since memory is becoming inexpensive.
- In many typical application environments, current computing resources are more than adequate to handle space requirements of your programs.
- However, for robotics and other embedded environments, it is still somewhat necessary for space to be considered particularly for limited compute platforms.
  - Volume or weight constraints, power constraints, etc. may make space usage a critical concern.

Time Complexity

- Time required to complete a task.
- In the examples below, which function would take more time?

```
// sum all values in an array
int sum (int *a, int size)
{
    int temp = 0, i;
    for (i=0; i<size; i++)
        temp += a[i];
    return temp;
}
```

```
// sum all values in an array (recursive)
int sum (int *a, int size)
{
    int temp = 0, i;
    for (i=0; i<size; i++)
        if (size == 0) return 0;
        temp = *a + sum (++a, --size);
    return temp;
}
```

Designing for Efficiency

- Usually there are choices in method used
- Circumstances may set limits on choices available
- May need to be very careful about choosing efficient methods

Size of Problem

- Example: Sort data records – size of problem is number of records (`n`). We may have no concerns about resources:
  - For a small `n`, any sorting method will be adequate.
  - For millions of `n`, choosing a sorting method becomes more important if time is of concern. If there is no deadline, any method would also work.
- When we look at algorithms, we are concerned about:
  - `worst_time(n)`, `average_time(n)`
  - `worst_space(n)`, `average_space(n)`
- In other words, we are interested in time and space complexity in terms of average and maximum data size and processing time.

Purpose of Analysis (1)

- Goal is to develop metrics that allows us to:
  - Choose best algorithm based on size of data
  - Predict behavior of algorithm when quantity of data increases

- How do we compare and contrast algorithms' performance in terms of time and space?
  - Exact measurements can be significantly affected by compiler, processor, language used, and different computers.
  - We need a more general way to compare algorithms in terms of time and space

Purpose of Analysis (2)

- We want a way of expressing relationship between size of problem and compute time.
- Our metric will be used to compare algorithms, however, this will not produce any exacting quantitative comparison.
- Given an algorithm:
  - We need to be able to describe these values mathematically
  - We need a systematic means of using the description of the algorithm together with the properties of an associated data structure
  - We need to do this in a machine-independent way

Time Complexity Example – Summing Array Elements

- Array with:
  - 1 element – 1 time units + overhead
  - 10 elements – 10 time units + overhead
  - 1000 elements – 1000 time units + overhead
- Time required is directly proportional to size/data in the array

```
// sum all values in an array
int sum (int *a, int size)
{
    int temp = 0, i;
    for (i=0; i<size; i++)
        temp += a[i];
    return temp;
}
```

```
worst_time ~ (fixed_overhead + n * time_units)
```

Time Complexity Example – Process all Elements in a 2D Array

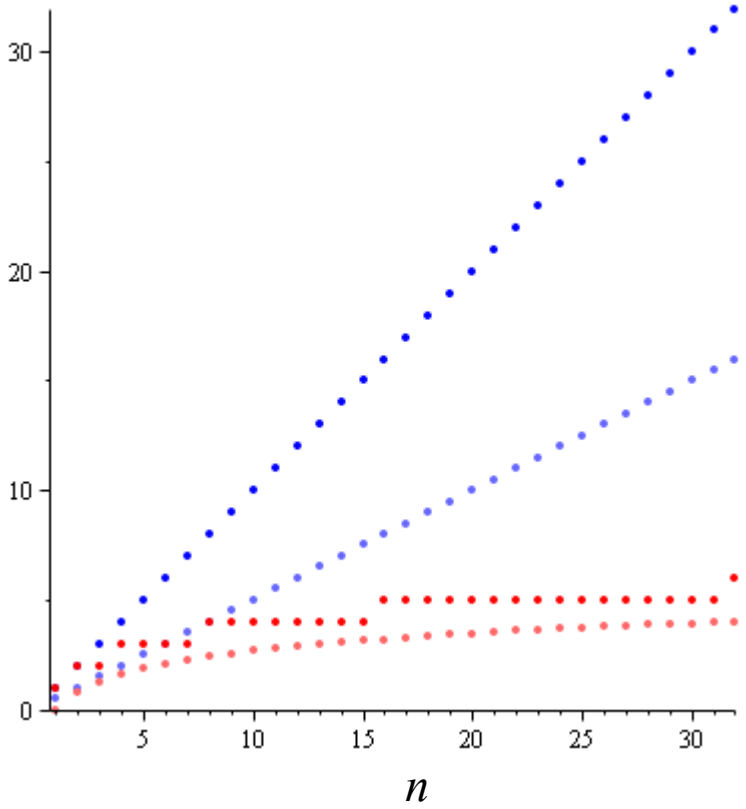
- Array with:
  - 1x1 element – 1 time unit
  - 10x50 Array – 500 time units
  - 400x60 Array – 24 000 time units
- Time varies as product of row and column size
  - General case: #row, #cols independent
  - Worst case: $(\max(r,c))^2$

```
// a is a 2D array of integers
int row, col;
for (row=0; row<size; row++)
    for (col=0; col<size; col++)
        process (a[row][col]);
```

```
worst_time ~ (fixed_overhead + n^2)
```

Example from a Future Topic (Sorting)

- There are other algorithms which are significantly faster as the problem size increases
- This plot shows maximum and average number of comparisons to find an entry in a sorted array of size *n*
  - Linear search
  - Binary search



Units of Work

- Because we are really only interested in complexity of an algorithm in terms of input data size, we don't address the issues of complexity or efficiency at a unit level (i.e., we don't care about the complexity of calculations)
- Examples:

```
// assignment
c = d;
```

```
// power
C = power(d*e + f*g, 2);
```

```
// sum all values in an array
int sum (int *a, int size)
{
     int temp = 0, i;
     for (i=0; i<size; i++)
          temp += a[i];
     return temp;
}
```

```
// sum all values in an array (recursive)
int sum (int *a, int size)
{
     int temp = 0, i;
     for (i=0; i<size; i++)
     if (size == 0) return 0;
     temp = *a + sum (++a, --size);
     return temp;
}
```

Asymptotic Analysis – Big-O Notation (1)

- Given an algorithm, we need:
  - to be able to describe these values mathematically
  - a systematic means of using the description of the algorithm together with the properties of an associated data structure
  - to do this in a machine-independent way
- For this, we use a notation called Big-O Notation
  - Also known as Bachmann-Landau Notation or Asymptotic Notation
  - The reason why it's called asymptotic notation will become apparent soon (hint: has something to do with the description of curves)

Asymptotic Analysis – Big-O Notation (2)

- Big-O notation gives us an approximation for how an algorithm execution time/space expands with problem/data size.
  - Example: How much longer will it take to process 1,000,000 things versus 100 things?
  - I like to think of big-O as meaning "on the order of"
- Big-O is used to describe worst-case performance of an algorithm.
  - There are other Bachmann-Landau symbols that can be used to describe best-case or average-case performance, but we won't be covering those in this course.

Common Complexity Classes (1)

Increasing Complexity

Adjective	Big-O Notation	Description
Constant	O(1)	Same length of space/time regardless of data size
Logarithmic	O(log n)	Grows as a logarithm of data size
Linear	O(n)	Grows at the same rate as data set size
n log n	O(n log n)	Grows as a logarithm of data size multiplied by data size
Quadratic	$O(n^2)$	Grows as a square of data size
Cubic	$O(n^3)$	Grows as a cubic of data size
Exponential	$O(2^n)$	Grows exponentially
Exponential	$O(10^n)$	Grows exponentially

Common Complexity Classes (2)

- O(1) – Pure overhead, algorithm doesn't grow with data

Increasing Complexity

Adjective	Big-O Notation	Description
Constant	O(1)	Same length of space/time regardless of data size
Logarithmic	O(log n)	Grows as a logarithm of data size
Linear	O(n)	Grows at the same rate as data set size
n log n	O(n log n)	Grows as a logarithm of data size multiplied by data size
Quadratic	$O(n^2)$	Grows as a square of data size
Cubic	$O(n^3)$	Grows as a cubic of data size
Exponential	$O(2^n)$	Grows exponentially
Exponential	$O(10^n)$	Grows exponentially

Common Complexity Classes (3)

- O(n) – We should expect the algorithm with O(n) complexity to take 10 times as long to process 1000 items as it does 100 items.

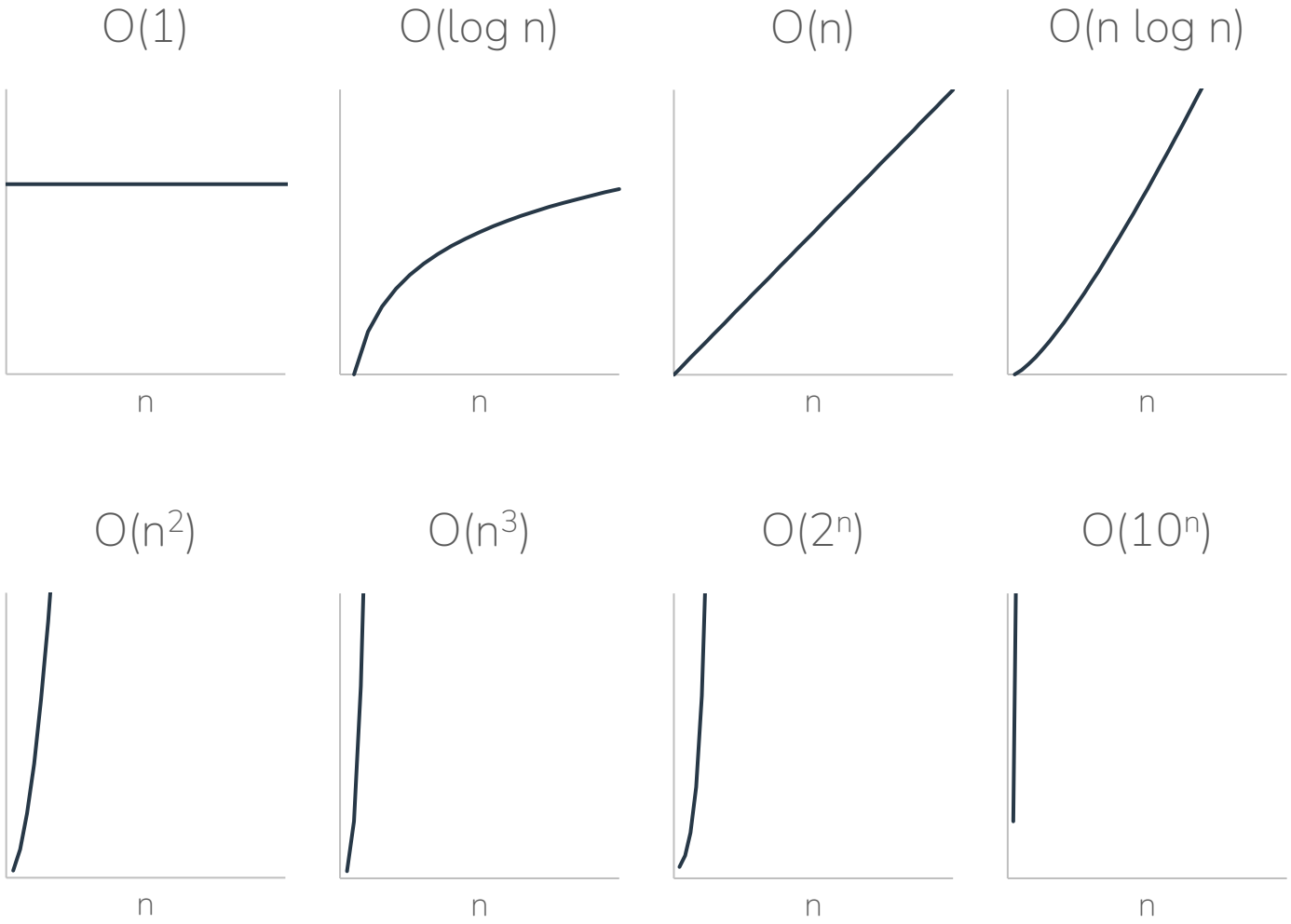
Increasing Complexity

Adjective	Big-O Notation	Description
Constant	O(1)	Same length of space/time regardless of data size
Logarithmic	O(log n)	Grows as a logarithm of data size
Linear	O(n)	Grows at the same rate as data set size
n log n	O(n log n)	Grows as a logarithm of data size multiplied by data size
Quadratic	$O(n^2)$	Grows as a square of data size
Cubic	$O(n^3)$	Grows as a cubic of data size
Exponential	$O(2^n)$	Grows exponentially
Exponential	$O(10^n)$	Grows exponentially

What is n?

- Consistent measure of size for the type of problem.
- Sorting, searching – size of record collection.
- Essentially, for a given problem, you will know what the data units are.

Graphical Display of Complexity

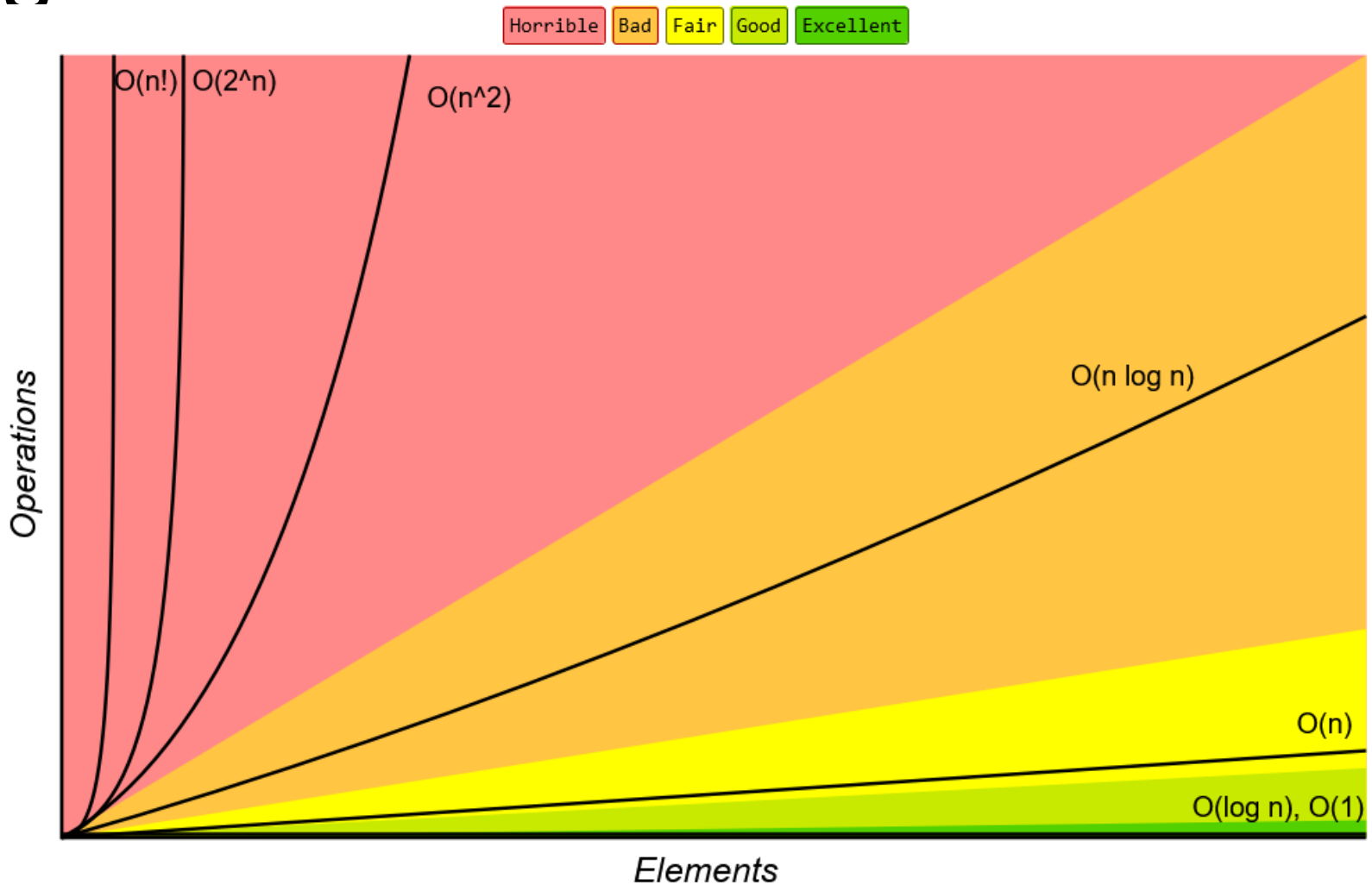


when you reduce the time complexity of your algorithm from O(n) to O(1)

The lord of time

The Big-O Cheatsheet

www.bigocheatsheet.com



In General…

$$f(n) = \text{coefficient}*(\text{dominant term}) \pm (\text{lesser terms})$$

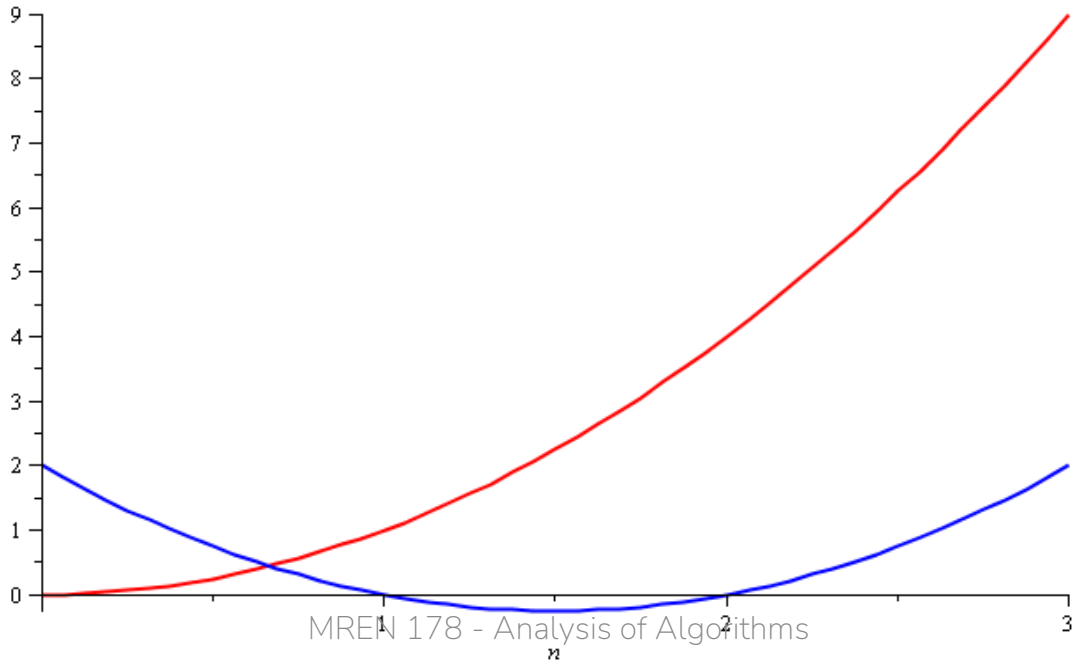
$$O(f(n)) = \cancel{\text{coefficient}}*((\text{dominant term}) \pm \cancel{(\text{lesser terms})}$$
$$= O(\text{dominant term})$$

Quadratic Growth Example (1)

- Consider the two functions

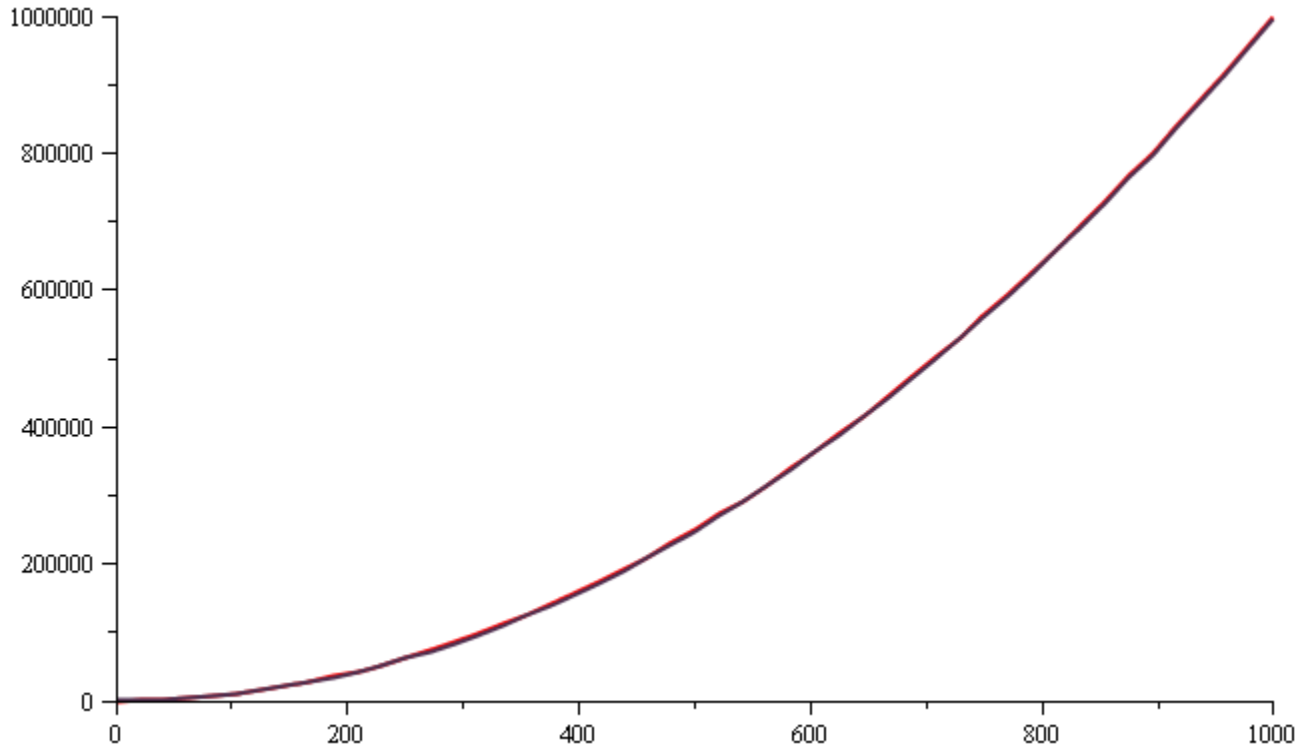
$$f(n) = n^2 \text{ and } g(n) = n^2 - 3n + 2$$

- Around $n = 0$, they look very different.



Quadratic Growth Example (2)

- Yet on the range **n** = [0, 1000], they are (relatively) indistinguishable:



$O(n^2)$

Quadratic Growth Example (3)

- The absolute difference is large, for example,

$$f(1000) = 1\,000\,000$$

$$g(1000) = \ 997\,002$$

but the relative difference is very small.

$$\left|\frac{f(1000) - g(1000)}{f(1000)}\right| = 0.002998 < 0.3\%$$

and this difference goes to zero as $n \to \infty$.

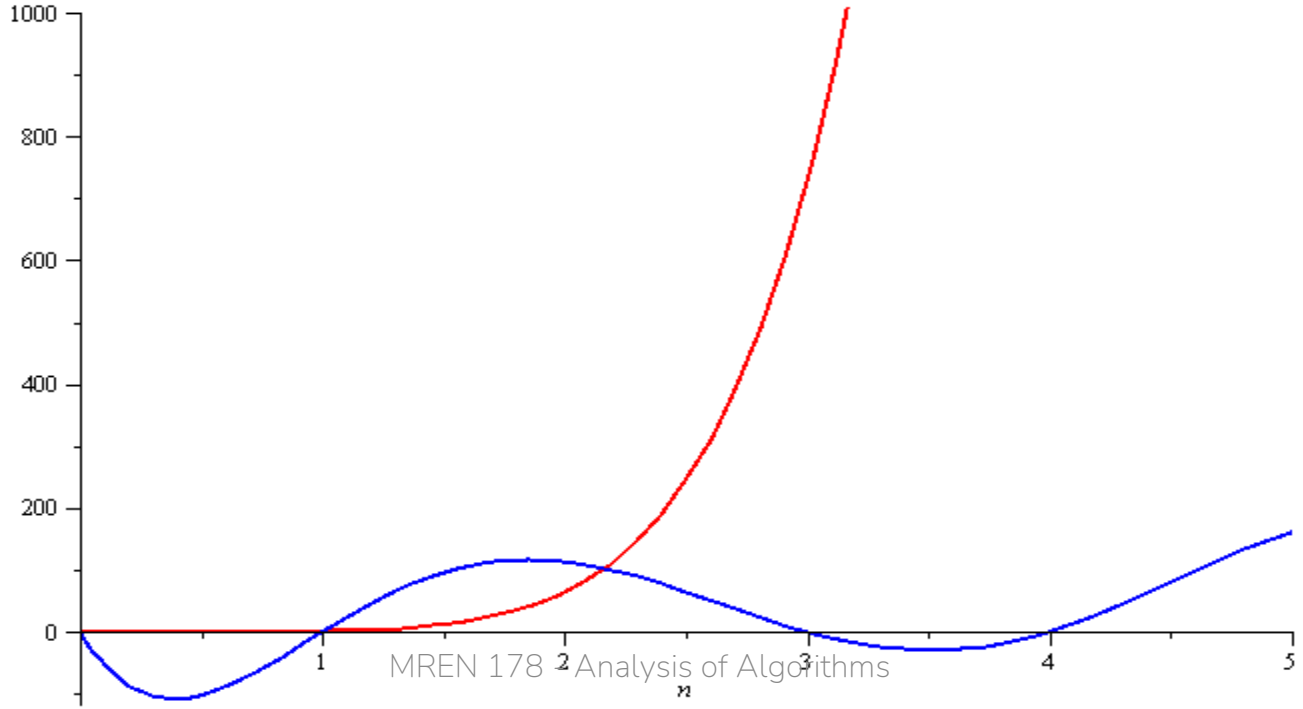
Both of these functions are $O(n^2)$

Polynomial Growth Example (1)

- To demonstrate with another example,

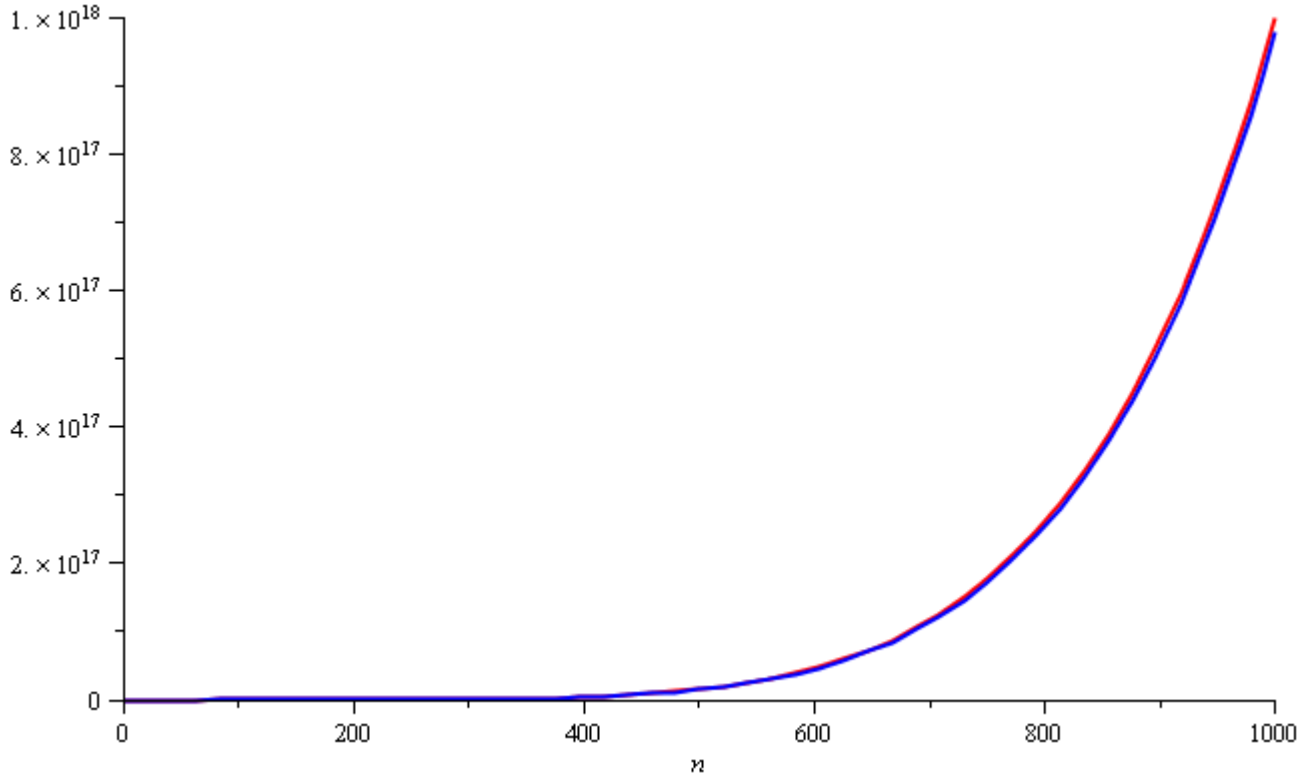
  $f(n) = n^6$ and $g(n) = n^6 - 23n^5 + 193n^4 - 729n^3 + 1206n^2 - 648n$

- Around $\mathbf{n = 0}$, they look very different.



Polynomial Growth Example (2)

- Around $\mathbf{n = 1000}$, the relative difference is less than 3%



$O(n^6)$

Polynomial Growth (3)

- The justification for both pairs of polynomials being similar is that, in both cases, they each had the same leading term:
  - $\mathbf{n^2}$ in the first (quadratic) case -> $O(n^2)$
  - $\mathbf{n^6}$ in the second (polynomial) case -> $O(n^6)$
  - We ignore contributions of lesser terms since they don't contribute much of the function's value as n increases.
- Suppose however, that the coefficients of the leading terms were different:
  - In this case, both functions would exhibit the same rate of growth, however, one would always be proportionally larger.
  - However, for our case, we also don't really care about absolute differences defined by coefficients; we care about growth behaviours of algorithms and can thus drop leading coefficients.

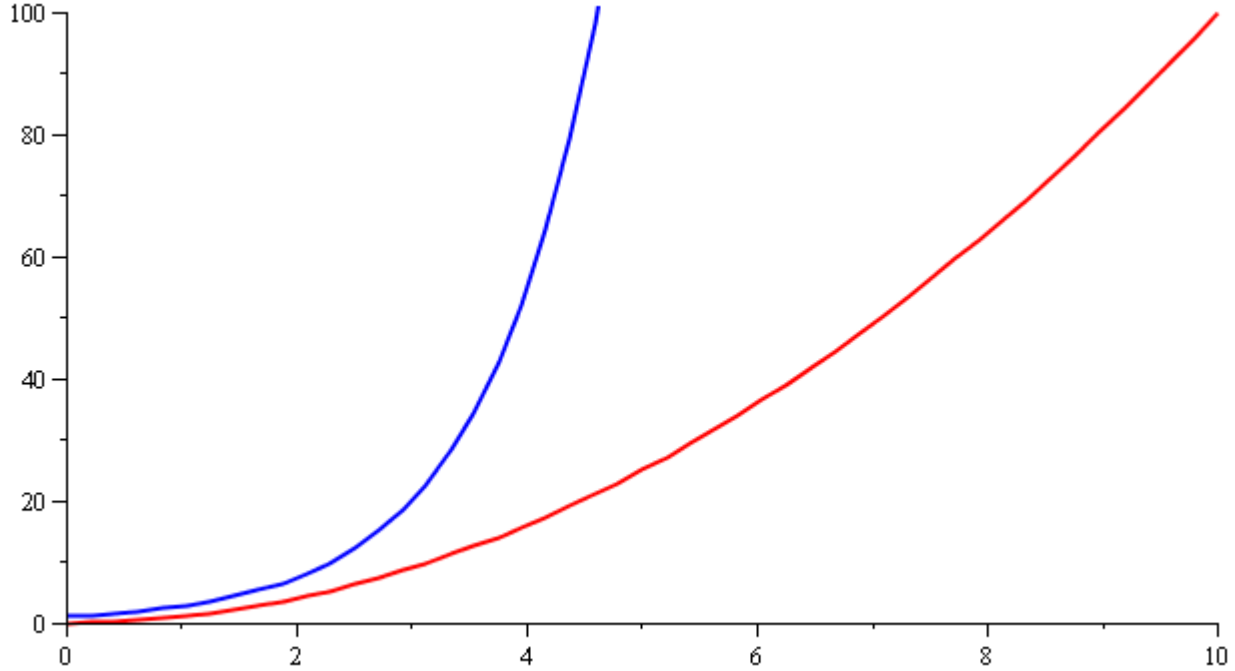
Algorithm Analysis (1)

- An algorithm is said to have polynomial time complexity if its run-time may be described by $O(n^d)$ for some fixed $d \geq 0$
  - We will consider such algorithms to be efficient

- Problems that have no known polynomial-time algorithms are said to be intractable
  - Traveling salesman problem: find the shortest path that visits n cities
  - Best run time: $O(n^2 2^n)$

Algorithm Analysis (2)

- In general, you don't want to implement exponential-time or exponential-memory algorithms
  - Warning: don't call a quadratic curve "exponential"

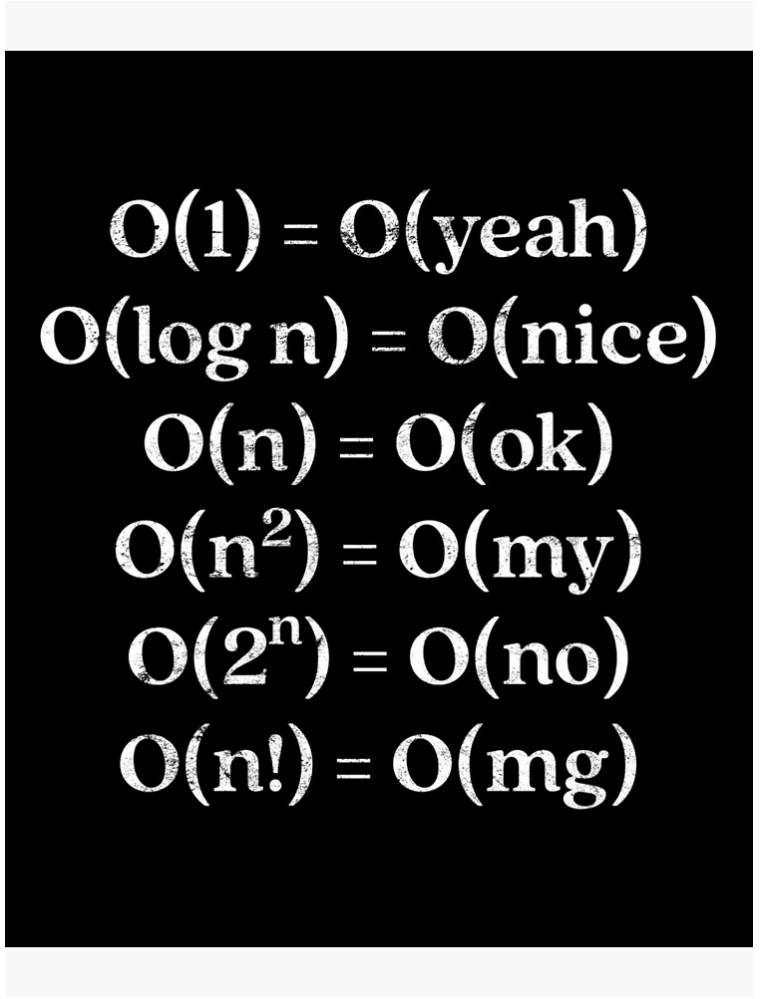


Simplified Calculations of Big-O

- We generally don't bother with in-depth calculations for worst/average time/space complexity since they are crude approximations anyway
  - Remember, they are dependent on things like compiler and computer
- Big-O gives feeling about how long something will take – an approximation of approximations
- The value of Big-O is being able to compare one method to another
  - There is no absolute value for O(n) – only a hierarchy.
- For upper bound – essentially worst-case behaviour.

Algorithm Analysis

- To properly investigate the determination of run times asymptotically, we will :
  - Begin with machine instructions
  - Discuss operations
  - Control statements
    - Conditional statements and loops
  - Functions
  - Recursive functions



Machine Instructions

- Given any processor, it is capable of performing only a limited number of operations.
- These operations are called instructions.
- The collection of instructions is called the instruction set.
  - The exact set of instructions differs between processors
  - MIPS, ARM, x86, 6800, 68k
- Any instruction runs in a fixed amount of time (an integral number of CPU cycles)

Operators

- Because each machine instruction can be executed in a fixed number of cycles, we may assume each operation requires a fixed number of cycles
- The time required for any operator is O(1), including:
  - Retrieving/storing variables from memory
  - Variable assignment `=`
  - Integer operations `+ - * / % ++ --`
  - Logical operations `&& || !`
  - Bitwise operations `& | ^ ~`
  - Relational operations `== != < <= => >`
  - Memory allocation and deallocation `malloc free`

Memory allocations and deallocations are actually really slow – they are the slowest by a significant factor (over 100x). (De)allocation may not be O(1), but for our purposes we consider these operations to be fixed time O(1).

Blocks of Operations

- Each operation runs in O(1) time and therefore any fixed number of operations also run in O(1) time, for example:

```
// swap variables a and b
int tmp = a;
a = b;
b = tmp;

// update a sequence of values
++index;
prev_modulus = modulus;
modulus = next_modulus;
next_modulus = modulus_table[index];
```

Blocks in Sequence (1)

- Essentially, look at code and count operations

```
int count_zeros (int *x, int len) {
    int count = 0;
    int i;
    for (i=0; i<len; i++)
        if (x[i]==0) count++;
    return count;
}
```

Remember: we don't need to count every single line

O(1) enter function overhead

O(1) declare/initialize variables

O(1) for initializing for loop

n*O(1) for loop compare

n*O(1) for loop add

n*O(1) compare

n*O(1) add (worst case)

O(1) return/exit overhead

O(4n+4) -> O(n)

Blocks in Sequence (2)

- Suppose you have now analyzed a number of blocks of code run in sequence

```
// increase array capacity by delta
void update_capacity (int *array, int array_size, int delta) {
    int *array_old = array;
    int size_old = array_size;
    array_size += delta;
    array = (int*) malloc(sizeof(int)*array_size);

    for ( int i = 0; i < size_old; ++i )
        array[i] = array_old[i];

    free(array_old);
}
```

O(1)

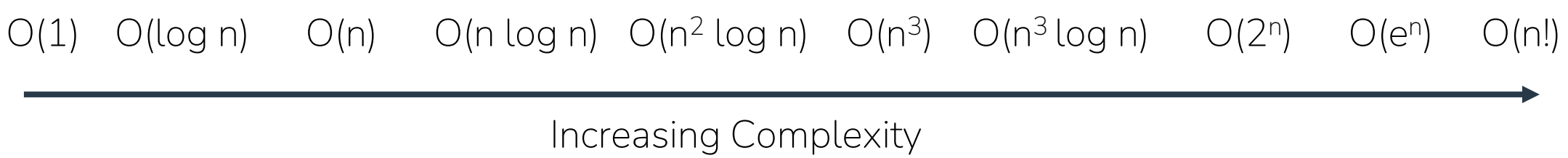
O(n)

O(1)

To calculate the total run time, add the entries: O(1 + n + 1) = O(n)

Blocks in Sequence (3)

- Other examples include:
  - Run three blocks of code which are $O(1)$, $O(n^2)$, and $O(n)$
    - Total run time $O(1 + n^2 + n) = O(n^2)$
  - Run two blocks of code which are $O(n \log n)$, and $O(n^{1.5})$
    - Total run time $O(n \log n + n^{1.5}) = Q(n^{1.5})$
- Linear ordering of complexity



- When considering a sum, take the most dominant term

Control Statements (1)

- These are statements which potentially alter the execution of instructions
  - Conditional statements – `if`, `switch`
  - Condition-controlled loops – `for`, `while`, `do-while`
- Given any collection of nested control statements, it is always necessary to work inside out
  - Determine the run times of the inner-most statements and work your way out

Control Statements (2)

- Given:

```
if( condition ) {
    // true body
} else {
    // false body
}
```

- The run time of a conditional statement is:
  - the run time of the condition (the test), plus
  - the run time of the body which is run
- In most cases, the run time of the condition is O(1)

Control Statements (3)

- In some cases, it is easy to determine which statement must be run:

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

Control Statements (4)

- In others, it is less obvious.
- In this case, we actually don't know how many times the highlighted statement will run
  - If the list is sorted in ascending order it will always be run
  - If the list is sorted in descending order it will never be run
  - If the list is uniformly randomly distributed, then ???

```
// finds max item in array
int find_max( int *array, int n ) {
    max = array[0];
    for ( int i = 1; i < n; ++i ) {
        if ( array[i] > max ) {
            max = array[i];
        }
    }
    return max;
}
```

Condition-Controlled Loops (1)

- The C for loop is a condition-controlled statement:

```
for (int i = 0; i < N; ++i) {// ...}
```

- is identical to:

```
int i = 0;              // initialization
while (i < N) {         // condition
    // …
    ++i;                // increment
}
```

- The initialization, condition, and increment usually are single statements running in O(1)
- Assuming there are no breaks or return statements in the loop, the run time for the loop is O(n)

Condition-Controlled Loops (2)

- What if the body does not depend on the variable (in this example, `i`)?

```
for (int i = 0; i < n; ++i) {
    // code that is O(f(n))
}
```

  - If the body is O(f(n)), then the run time of the loop is O(n f(n)).
- For example:

```
int sum = 0;
for (int i = 0; i < n; ++i) {
    sum += 1; // O(1)
}
```

  - This code has run time O(n*1) = O(n)

Condition-Controlled Loops (3)

- Another example:

```
int sum = 0;
for ( int i = 0; i < n; ++i ) {
    for ( int j = 0; j < n; ++j ) {
        sum += 1;      // O(1)
    }
}
```

  - The previous example showed that the inner loop is O(n), thus the outer loop is O(n*n) = $O(n^2)$

Analysis of Repetition Statements

- Suppose with each loop, we use a linear search an array of size `m`:

```
for ( int i = 0; i < n; ++i ) {
    // search through an array of size m
    // O(m)
}
```

  - The inner loops is O(m) and thus the outer loop is O(n m) -> $O(n^2)$

Conditional Statement

- Consider this example:

```
int clr_fcn(…) {
    if (sets == n) {
        return;
    }

    max_height = 0;
    num_disjoint_sets = n;

    for ( int i = 0; i < n; ++i )
{
        parent[i] = i;
        tree_height[i] = 0;
    }
}
```

O(1)

O(1)

O(n)

O(1)

If sets == n, then runtime is O(1)
Otherwise, runtime is O(n)

Analysis of Repetition Statements (1)

- If the body does depends on the variable (in this example, `i`), then the run time of

```
for ( int i = 0; i < n; ++i ) {
    // code with runtime O(f(i,n))
}
```

  - is

$$O\left(1+\sum_{i=0}^{n-1}\left(1+f(i,n)\right)\right)=O\left(1+n+\sum_{i=0}^{n-1}f(i,n)\right)$$

Analysis of Repetition Statements (2)

```
int sum = 0;
for ( int i = 0; i < n; ++i ) {
    for ( int j = 0; j < i; ++j ) {
        sum += i + j;
    }
}
```

- The inner loop is O(1+i(1+1)) = O(i), hence the outer loop is:

$$O\left(1+\sum_{i=0}^{n-1}(1+i)\right)=O\left(1+n+\sum_{i=0}^{n-1}i\right)=O\left(1+n+\frac{n(n-1)}{2}\right)=O(n^2)$$

Analysis of Repetition Statements

```
int sum = 0;
for ( int i = 0; i < n; ++i ) {
    for ( int j = 0; j < i; ++j ) {
        for ( int k = 0; k < j; ++k ) {
                sum += i + j + k;
        }
    }
}
```

From inside to out:

$O(1) \rightarrow O(j) \rightarrow O(i^2) \rightarrow O(n^3)$

Control Statements

- Switch statements are nested if statements:

```
switch( i ) {
    case 1:   /* do stuff */ break;
    case 2:   /* do other stuff */ break;
    case 3:   /* do even more stuff */ break;
    case 4:   /* well, do stuff */ break;
    case 5:   /* tired yet? */ break;
    default:  /* do default stuff */
}
```

- They run in O(n) time where n is the number of cases.

Serial Statements

- Suppose we run one block of code followed by another block of code, such code is said to be run serially.
- If the first block of code is O(f(n)) and the second is O(g(n)), then the run time of two blocks of code is:

$$O(f(n) + g(n))$$

- This usually (for algorithms not including function calls) simplifies to one or the other.
- For example:
  - $O(n) + O(n^2) + O(n^4) = O(n + n^2 + n^4) = O(n^4)$

Functions (1)

- A function (or subroutine) is code which has been separated out, either to:
  - Perform repeated operations (e.g., mathematical functions)
  - Group related tasks (e.g., initialization)
- Because a function can be called from anywhere, we must:
  - prepare the appropriate environment
  - deal with arguments (parameters)
  - jump to the subroutine
  - execute the subroutine
  - deal with the return value
  - clean up

Functions (2)

- Fortunately, this is such a common task that all modern processors have instructions that perform most of these steps in one instruction
- Thus, we will assume that the overhead required to make a function call and to return is O(1)

Complexity Summary

- The goal of algorithm analysis is to take a block of code and determine the asymptotic run time or asymptotic memory requirements based on various parameters
- We are computing a metric that allows us to compare two methods and their change in behavior due to changes in problem size.
- Full timing analysis is much harder.
  - Example: algorithm for malloc() and free() may take variable lengths of time depending on what has happened prior. Thus, the overall program behavior cannot be predicted precisely if program uses malloc() and free().
  - If we are concerned with precise timing (e.g., real-time system with tight deadlines), we would explore how the compiler translates and optimizes code, and how library routines work.

We will be considering complexity each time we inspect any algorithm used in this class

Example Calculation for Nested Loops

- Inner loop - O(B)
  - loop operations performed B times
- Outer loop causes inner loop to be run A times - O(A*B)
- If A=B=n - $O(n^2)$
  - Even when A != B – worst case is A==B, so this is considered $O(n^2)$
- Note: in this case, the problem size is the dimensions of the array (A,B)

```
int nested_loop_fcn (...) {
    for (int i=0; i<A; i++) {
        for (int k=0; k<B; k++) {
            some work;
        }
    }
}
```

Example Calculation for Recursion

- Basically, recursion can be thought of like a loop (which it is…sort of)
- In the factorial code:
  - `if(x==0)` check is done every time
  - perform multiplication and call `factorial n-1` times – total of n operations
  - Thus, O(n)
  - Note: in this case, the problem size is the value of n.

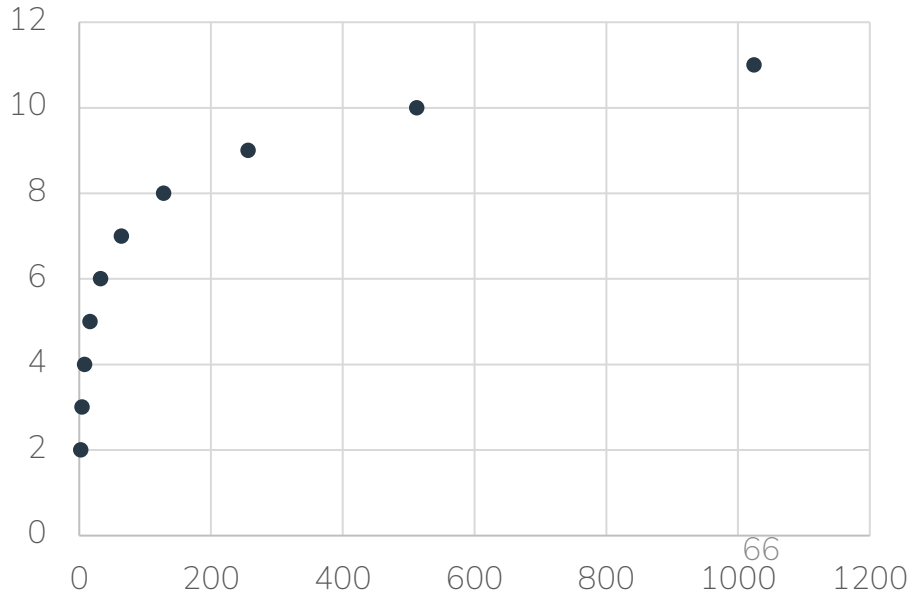
```
int factorial (x) {
    if(x==0) return 1;
    return x * fact(x-1);
}
```

Example Calculation

```
for (int i=n; i>=1; i=i/2)
    do_stuff();
```

- Let's walk through this with a few values of n
  - n=2 2, 1 (2 times)
  - n=4 4, 2, 1 (3 times)
  - n=16 16, 8, 4, 2, 1 (5 times)
  - n=32 32, 16, 8, 4, 2, 1 (6 times)
  - n=1024 1024, 512, 256, 128, 64, 32, 16, 8, 4, 2, 1 (11 times)

- $O(\log_2 n + 1)$
- $O(\log n)$



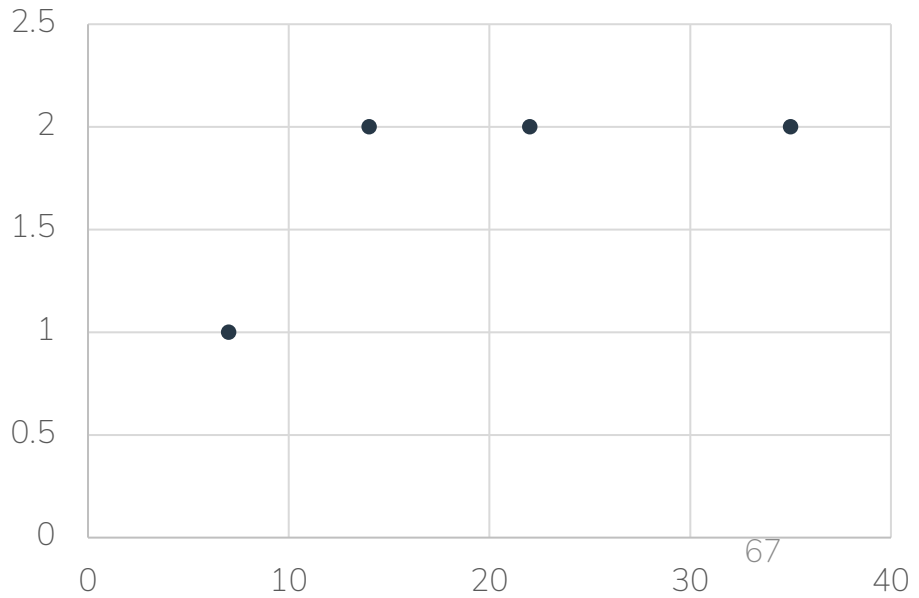
Example Calculation

```
for (int i=n; i>=1; i=i/7)
    do_stuff();
```

- Let's walk through this with a few values of n
  - n=7 7 (1 time)
  - n=14 14, 2 (2 times)
  - n=22 22, 3 (2 times)
  - n=35 35, 5 (2 times)

- $O(\log_7 n)$
- $O(\log n)$

All logarithmic growth is approximately the same – # of times is smaller than n.



Example Problem 1 (1)

- Suppose you have the following functions with information about worst-case performance, where `w` is a constant:

```
int f(int n, int w){
    do_work(); // work done is w*2 time units.
    return w*n/2;
}
void g(int n, int w) {
    do_more_work(); // work done is n*w time units in worst case.
}
```

- Consider the following code:

```
for (i=0; i<f(n, 10); i++)
    g(n, 20)
```

- What is complexity (Big-O) of this code?

Example Problem 1 (2)

```
f(n, w){
    do_work(); // work done is w*2 time units.
    return w*n/2;
}
g(n, w) {
    do_more_work(); // work done is n*w time units in worst case.
}
```

```
for (i=0; i<f(n, 10); i++)
    g(n, 20)
```

Component	Complexity
Loop body	O(20n)
For loop initialization	O(1)
For loop compare	O(20)
For loop increment	O(1)
For loop repeats 5n times	5n

$O(1)+5n*(O(20)+O(1)+O(20n))$

$=O(1+100n+5n+100n^2)$

$=O(n^2)$

Example Problem 2 (1) - Definition

Consider the C program fragments given. Assume that `m`, `n` and `k` are unsigned `ints`, and that the functions `f_1`, `f_2`, `f_3`, and `f_4` have the following characteristics:

1. The worst running time for `f_1(n, m)` is O(1), and it returns a value between `1` and `(n+m)`.
2. The worst running time for `f_2(n, m, k)` is O(`nm + k`)
3. The worst running time for `f_3(n, m, k)` is O(`mk`)
4. The worst running time for `f_4(n, m, k)` is O(`n+k`)

Determine a tight Big-O expression for the worst-case running time of each of the following program fragments. Note that each line has a unique line number and assume the variable c is an arbitrary integer value and not a constant.

```
0:  // Fragment A.
1:  c = f_1(a, b)
2:
3:  // Fragment B.
4:  for (a = 0; a < c; a++)
5:     f_2(a, a, c);
6:
7:  // Fragment C.
8:  for (a = 0; a < f_1(0, c); a++)
9:     f_3(a, b, c);
10:
11: // Fragment D.
12: for (a = 0; a < c; a++)
13:    for (b = 0; b < a; b++)
14:       f_4(a, b, c);
```

Example Problem 2 (2) – Fragment A

Consider the C program fragments given. Assume that `m`, `n` and `k` are unsigned `ints`, and that the functions `f_1`, `f_2`, `f_3`, and `f_4` have the following characteristics:

1. The worst running time for `f_1(n, m)` is O(1), and it returns a value between `1` and `(n+m)`.
2. The worst running time for `f_2(n, m, k)` is O(`nm + k`)
3. The worst running time for `f_3(n, m, k)` is O(`mk`)
4. The worst running time for `f_4(n, m, k)` is O(`n+k`)

Determine a tight Big-O expression for the worst-case running time of each of the following program fragments. Note that each line has a unique line number and assume the variable c is an arbitrary integer value and not a constant.

```
0:  // Fragment A.
1:  c = f_1(a, b)
2:
3:  // Fragment B.
4:  for (a = 0; a < c; a++)
5:     f_2(a, a, c);
6:
7:  // Fragment C.
8:  for (a = 0; a < f_1(0, c); a++)
9:     f_3(a, b, c);
10:
11: // Fragment D.
12: for (a = 0; a < c; a++)
13:    for (b = 0; b < a; b++)
14:       f_4(a, b, c);
```

O(1) – from statement 1

Example Problem 2 (3) – Fragment B

Consider the C program fragments given. Assume that `m`, `n` and `k` are unsigned `ints`, and that the functions `f_1`, `f_2`, `f_3`, and `f_4` have the following characteristics:

1. The worst running time for `f_1(n, m)` is O(1), and it returns a value between 1 and `(n+m)`.
2. The worst running time for `f_2(n, m, k)` is O(nm + k)
3. The worst running time for `f_3(n, m, k)` is O(mk)
4. The worst running time for `f_4(n, m, k)` is O(n+k)

Determine a tight Big-O expression for the worst-case running time of each of the following program fragments. Note that each line has a unique line number and assume the variable c is an arbitrary integer value and not a constant.

```
0:  // Fragment A.
1:  c = f_1(a, b)
2:
3:  // Fragment B.
4:  for (a = 0; a < c; a++)
5:     f_2(a, a, c);
6:
7:  // Fragment C.
8:  for (a = 0; a < f_1(0, c); a++)
9:     f_3(a, b, c);
10:
11: // Fragment D.
12: for (a = 0; a < c; a++)
13:    for (b = 0; b < a; b++)
14:       f_4(a, b, c);
```

- `f_2()` is O(aa + c)
- Loop is O(c) where c = a+b in the worst case
- O((aa+c)*(a+b)) or $O(n^3)$

Example Problem 2 (4) – Fragment C

Consider the C program fragments given. Assume that `m`, `n` and `k` are unsigned `ints`, and that the functions `f_1`, `f_2`, `f_3`, and `f_4` have the following characteristics:

1. The worst running time for `f_1(n, m)` is O(1), and it returns a value between 1 and `(n+m)`.
2. The worst running time for `f_2(n, m, k)` is O(nm + k)
3. The worst running time for `f_3(n, m, k)` is O(mk)
4. The worst running time for `f_4(n, m, k)` is O(n+k)

Determine a tight Big-O expression for the worst-case running time of each of the following program fragments. Note that each line has a unique line number and assume the variable c is an arbitrary

```
0:  // Fragment A.
1:  c = f_1(a, b)
2:
3:  // Fragment B.
4:  for (a = 0; a < c; a++)
5:      f_2(a, a, c);
6:
7:  // Fragment C.
8:  for (a = 0; a < f_1(0, c); a++)
9:      f_3(a, b, c);
10:
11: // Fragment D.
12: for (a = 0; a < c; a++)
13:     for (b = 0; b < a; b++)
```

- `f_3()` is O(bc)
- loop – `f_1()` is O(1). Calculates same value each call, so loop done (0+c) times in worst case. O(c)
- Total: O(cbc) ~ $O(n^3)$

Example Problem 2 (5) – Fragment D

Consider the C program fragments given. Assume that `m`, `n` and `k` are unsigned `ints`, and that the functions `f_1`, `f_2`, `f_3`, and `f_4` have the following characteristics:

1. The worst running time for `f_1(n, m)` is O(1), and it returns a value between `1` and `(n+m)`.
2. The worst running time for `f_2(n, m, k)` is O(`nm + k`)
3. The worst running time for `f_3(n, m, k)` is O(`mk`)
4. The worst running time for `f_4(n, m, k)` is O(`n+k`)

Determine a tight Big-O expression for the worst-case running time of each of the following program fragments. Note that each line has a unique line number and assume the variable c is an arbitrary

```
0:  // Fragment A.
1:  c = f_1(a, b)
2:
3:  // Fragment B.
4:  for (a = 0; a < c; a++)
5:     f_2(a, a, c);
6:
7:  // Fragment C.
8:  for (a = 0; a < F_1(0, c); a++)
9:     f_3(a, b, c);
10:
11: // Fragment D.
12: for (a = 0; a < c; a++)
13:    for (b = 0; b < a; b++)
14:       f_4(a, b, c);
```

- `f_4()` is O(a+c)
- Outer loop is done c times ~ O(c)
  - Inner loop is done a times, which is dependent on a used in outer loop
  - So inner loop is actually done 0, 1, 2, ..., c times - algebraic series sum = 0.5*c*(0+c) ~ $O(0.5c^2)$
- Total: $O(0.5c^2)*O(a+c) \sim O(n^3)$

Summary

- Big-O gives a feeling about how long something will take – an approximation of approximations
- The value of Big-O is that you can compare O() for one method to O() for another – no absolute value for O(n).
- Upper bound essentially represents worst-case behavior
- We will be considering complexity in terms of Big-O when we inspect any algorithm used in this class

Discussion

- Some problems just grow quickly with problem size, e.g., multiplying matrices.
- Some solutions may be able to use intermediate results – and not grow as rapidly as the problem size. Searching for values in an unsorted array versus searching in a sorted array. Worst case unsorted O(N); sorted O(logN).