GAME2001 Data Structures and Algorithms

Fall 2020

Week 1
Introduction
Big-O
Custom Data Structures vs STL

Introduction

- Data structures
  - building blocks of game development
  - not that useful by themselves
    - but combined with algorithms they become extremely powerful

DATA STRUCTURES

- defines how data is arranged in memory
- can be operated on by using various algorithms

- Array
  - defines how data is arranged in memory
    - variables or objects of a specific type
  - can be operated on by various algorithms
    - insertion into the array, deletion, searching, sorting

DATA STRUCTURES

- Arrays
- Link lists
- Queues
- Stacks
- Heaps
- Graphs
- Scene graphs
- Octrees
- Quad-trees
- Binary trees
- Minimax trees
- kd trees
- Sphere trees
- Bounding volume hierarchies
- Hash tables
- Portals and sectors

DATA STRUCTURES

- Another way to look at a data structure
  - a structure that is often user-defined and represents some kind of physical object.

- Data structures are also objects that contain objects.

ALGORITHMS

- code that manipulates data in data structures
  - inserting items into a data structure, deleting items, sorting, and iterating

- Recursion
- Insertions
- Deletions
- Merging
- Sorting algorithms
- Searching algorithms
- Transversal
- Balancing algorithms
- Data compression
- Texture compression
- Data encryption
- Texture filters

DATA STRUCTURES IN GAMES

- Topics we will be addressing:
  - How to store data efficiently in the computer's memory
  - How to process data efficiently
  - What algorithms work best under what situations and why
  - How various popular algorithms compare to one another in specific situations
  - What data structures can help the processing of game data in certain situations

DATA STRUCTURES IN GAMES

- Scene Management
  - Virtual environments more complex than hardware can handle
  - Thousands of polygons rendered, special effects are updated and drawn and physics processed. Can overwhelm a system. ‘
  - **Data structures** used since the beginning of 3D games to speed up rendering of scenes
    - Otherwise performance would be down to a noninteractive rate.
  - Examples:
    - Determining which geometry is visible and only rendering that
    - Avoiding or minimizing bottlenecks during state changes in the system
    - Managing static and dynamic objects

DATA STRUCTURES IN GAMES

- Artificial Intelligence
  - Drives many gaming experiences. Can be simple or complex.
  - Data structures used to control behavior of dynamic game elements.
  - More complex AI becomes an issue of speed. Higher complexity means longer processing times to complete its operations.
    - Combine that with multiple objects needing to execute AI algorithms.
  - Often find yourself balancing realism vs performance
  - AI in games often not as complex as AI used in robotics and other similar technologies.

DATA STRUCTURES IN GAMES

- Physics Dynamics and Collisions
  - Standard feature in 3D games
  - Represents how game objects interacts with the environment.
    - Forces (gravity, wind, etc..)
    - Collisions (and what happens after 2 things collide)
  - Will have their own specific data structures and algorithms specific to physics (point masses, rigid bodies, etc)
  - Physics so expensive to calculate, other data structures and algorithms for scene managements are also used in physics to help eliminate unnecessary calculations.

The C++ STL

- a standard set of template classes
- used for both code reuse and maintainability
- composed of many efficiently designed components and algorithms
- composed of containers, iterators, and algorithms
  - containers are the template classes that represent a data structure
  - iterator is used to transverse through a data structure
- main goal
  - to be abstract while maintaining efficiency

The C++ STL

- vector
- queue
- priority_queue
- list
- deque
- stack
- map
- multimap
- set
- multiset
- hash_set
- hash_multiset
- hash_map
- hash_multimap

- Part of C++ standard
  - bitset, string, valarray

Custom Data Structured vs STL

- STL Advantages
  - The STL is standard.
  - The STL is widely used.
  - The STL is optimized and efficient.
  - Using the STL saves time and effort.
  - The STL comes with C++.

Custom Data Structured vs STL

- STL Disadvantages
  - Debugging
  - Compiler error messages
  - Memory allocation
  - Code bloat (templates)
  - Compile times (templates)

Custom Data Structured vs STL

- Tools
  - Can almost always use STL
- Game code
  - Start with STL
  - Switch to custom data structures if STL cannot do the job

Templates

- a way to create code that is evaluated during compile time rather than runtime
- allow for the same code to be used with different data and user-defined types
  - minimizes the need to write duplicate code
- Function templates
  - allow programmers to specify function parameters to the template
- Class templates
  - allows programmers to specify the type for the template

Template Example

```
#include<iostream>
using namespace std;

template<class T>
T min_func(T lVal, T rVal)
{
    if(lVal > rVal)
        return rVal;
    return lVal;
}

template<class T>
T max_func(T lVal, T rVal)
{
    if(lVal < rVal)
        return rVal;
    return lVal;
}
```

```
template<class T>
class TemplateClass
{
    public:
        TemplateClass(T val)
        {
            m_val = val;
        }

        bool operator<(TemplateClass &rVal)
        {
            return m_val < rVal.GetVal();
        }

        bool operator>(TemplateClass &rVal)
        {
            return m_val > rVal.GetVal();
        }

        T GetVal()
        {
            return m_val;
        }
    private:
        T m_val;
};
```

Template Example

```
47  int main(int args, char **argc)
48  {
49      cout << "Template Example" << endl << endl;
50
51      cout << "Min = " << min_func(32, 54) << endl;
52      cout << "Max = " << max_func(49.3, 38.98) << endl;
53      cout << "Max (objects) = "
54           << max_func(TemplateClass<int>(7),
55                       TemplateClass<int>(4)).GetVal()
56           << endl;
57
58      TemplateClass<int> obj(10);
59      cout << "obj = " << obj.GetVal() << endl;
60      cout << endl << endl;
61
62      return 1;
63  }
```

Template Drawbacks

- Complexity
  - Difficult to read
  - Difficult to modify and update
  - Difficult to debug
- Dependencies
  - Increased coupling and compile times since the definition has to be with the declaration
- Code Bloat
  - Since it creates the class and functions at compile time for each type
- Compiler Support
  - Different compilers have different type of template support

Big-O Notation

- used to describe the theoretical performance of an algorithm
- measures the time or memory consumption used by an algorithm
- create a representation of the performance of an algorithm to compare it to others

Big-O Notation

- Problem
  - performance of algorithms can fluctuate depending on the number of items that were inserted and are present in a data structure
  - Example: if algorithm A runs faster than algorithm B with 1,000 items but runs slower with 100,000 items, it can be hard to get an accurate picture when comparing the two

- Big-O notation gives an idea of an algorithm's performance based on the number of items in the structure (its growth).

Big-O Notation

- Example
  - Line 3 and 8 (2)
    - Count for one unit each
  - Line 6 (4N)
    - Four units per time executed
      - 2 multiplications
      - 1 addition
      - 1 assignment
    - And is executed N times
  - Line 4 (2N + 2)
    - Initializing (1)
    - Comparing (N + 1)
    - Incrementing (N)
  - Total = 6N + 4

```
1  int sum( int n )
2  {
3      int partialSum = 0;
4      for(int i = 1; i <= n; ++i)
5      {
6          partialSum += i * i * i;
7      }
8      return partialSum;
9  }
```

Big-O Notation

- Big-O notation is identified as the term in the function that increases fastest relative to the size of the problem
  - $N^4 + 100N^2 + 10N + 50$
    - Big-O notation = $O(N^4)$
    - Other terms can be ignored
      - They are insignificant as N gets larger

Big-O Notation

- O(1)
  - Constant or bounded time
  - Assigning a value to the i^{th} element in an array of N elements
- $O(\log_2 N)$
  - Logarithmic time
  - Algorithms that successfully cut the amount of data in half typically fall into this category
- O(N)
  - Linear time
  - Printing all elements is a list of N elements

Big-O Notation

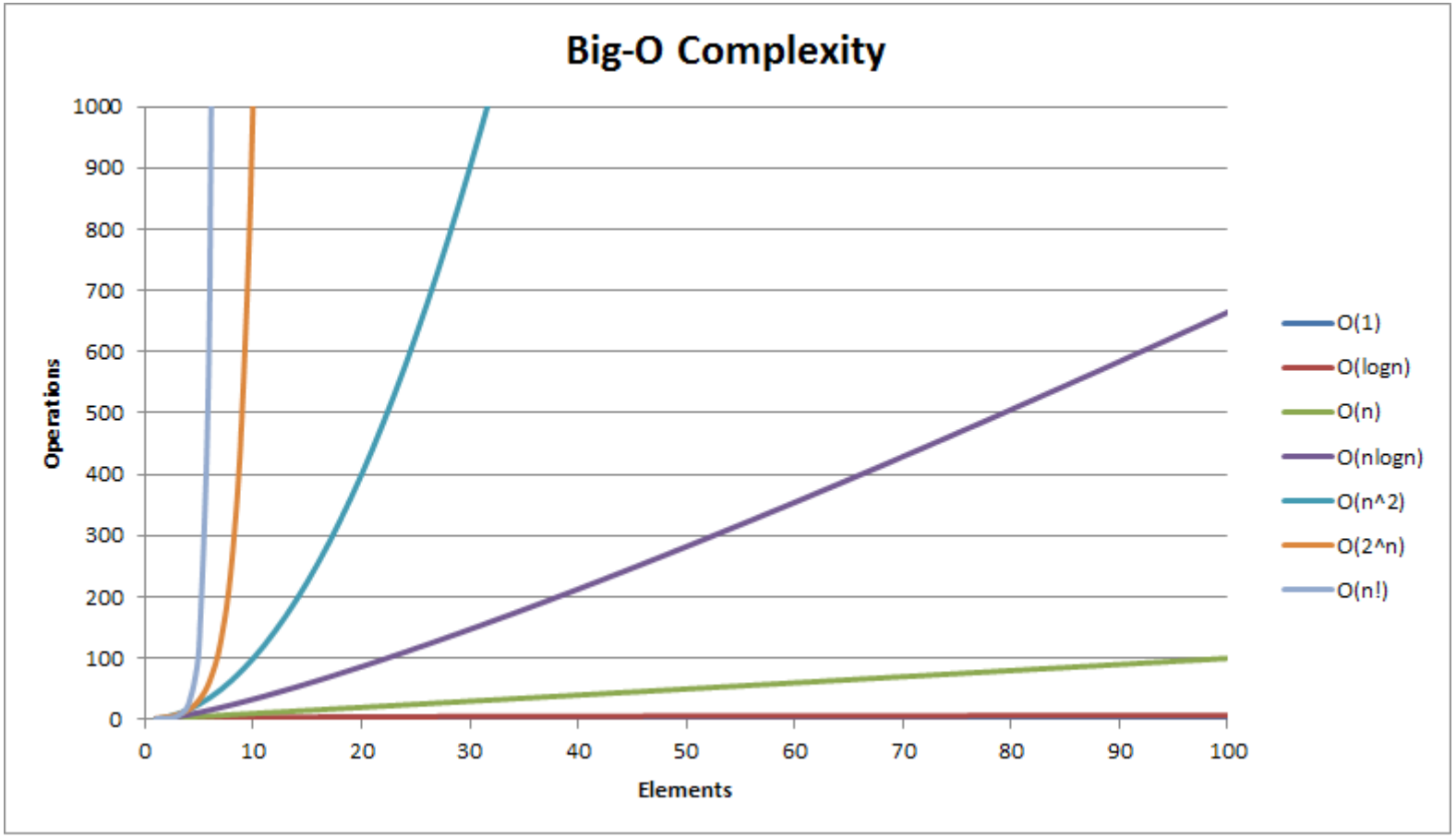
- $O(N \log_2 N)$
  - $N \log_2 N$ time
  - Applying a logarithmic algorithm N times
    - better sorting algorithms (Quicksort, Heapsort, Merge sort)
- $O(N^2)$
  - Quadratic time
  - Applying a linear algorithm N times
  - Simple sorting algorithms (Bubble sort, Insertion sort)

Big-O Notation

- $O(N^3)$
  - Cubic time
  - A routine that increments every element in a three dimensional table of integers
- $O(2^N)$
  - Exponential time
  - VERY COSTLY, increases exponentially

Big-O Notation

N	$\log_2 N$	$N \log_2 N$	N^2	N^3	2^N
1	0	1	1	1	2
2	1	2	4	8	4
4	2	8	16	64	16
8	3	24	64	512	256
16	4	64	256	4,096	65,536
32	5	160	1,024	32,768	4,294,967,296
64	6	384	4,096	262,114	1.8×10^{19}
128	7	896	16,384	2,097,152	3.4×10^{38}
256	8	2,048	65,536	16,777,216	1.1×10^{77}


Big-O Complexity
Operations
1000
900
800
700
600
500
400
300
200
100
0
0
10
20
30
40
50
60
70
80
90
100
Elements
O(1)
O(logn)
O(n)
O(nlogn)
O(n^2)
O(2^n)
O(n!)

Big-O Notation

- Example
  - Calculate the sum of integers from 1 to N

```
1  int sum( int n )
2  {
3      int sum = 0;
4      for(int count = 1; count <= n; ++count)
5      {
6          sum += count;
7      }
8      return sum;
9  }
```

  - What is Big-O?
  - Is there a better algorithm?

Big-O Notation

- Consider the following calculation when N = 5

```
  1  + 2  + 3  + 4  + 5
+ 5  + 4  + 3  + 2  + 1
-----------------------------------
  6  + 6  + 6  + 6  + 6  = 6 * 5 = 30
```

- We pair up each number from 1 to N with another, such that each pair adds up to N + 1
- There are N such pairs, giving us a total of (N+1)*N
- Since each number is included twice, we divide the product by 2
  - [(5+1)*5]/2 = 15

Big-O Notation

- Example
  - Calculate the sum of integers from 1 to N
    - Using formula

```
1  int sum( int n )
2  {
3      return ((n+1)*n)/2;
4  }
```

  - What is Big-O?
  - Is this always faster?

Big-O Notation

- General Rules
  - Rule 1 – FOR loops
    - Running time of the statements of the for loop times the number of iterations
  - Rule 2 – Nested loops
    - Analyze these inside out
    - Running time of the statement multiplied by the product of the sizes of all the loops
      - $O(n^2)$

```
1 for( int i = 0; i < n; ++i)
2     for( int j = 0; j < n; ++j)
3         ++k;
```

Big-O Notation

- Rule 3 – Consecutive Statements
  - These just add
    - $O(n^2)$

```
1  for( int i = 0; i < n; ++i)
2      a[i] = 0;
3  for( int i = 0; i < n; ++i)
4      for( int j = 0; j < n; ++j)
5          a[i] += a[j] + i + j;
```

- Rule 4 – If/Else
  - Never more than the running time of the test plus the larger of the running times of statement 1 and statement2

```
1  if (condition)
2      statement1
3  else
4      statement2
```

  - Can be overestimated in some cases, but never underestimated

Big-O Notation

```
#include <time.h>
#include <iostream>
using namespace std;

const int n = 500;
float TestData[n][n][n];

double diffclock(clock_t clock1,clock_t clock2)
{
    double diffticks=clock1-clock2;
    double diffms=(diffticks*10)/CLOCKS_PER_SEC;
    return diffms;
}

void column_ordered()
{
    for (int k = 0; k < n; ++k)
        for (int j = 0; j < n; ++j)
            for (int i = 0; i < n; ++i)
                TestData[i][j][k] = 0;
}

void row_ordered()
{
    for (int i = 0; i < n; ++i)
        for (int j = 0; j < n; ++j)
            for (int k = 0; k < n; ++k)
                TestData[i][j][k] = 0;
}
```

- Not the only thing we need to worry about

```
int main()
{
    clock_t begin=clock();
    row_ordered();
    clock_t end=clock();
    cout << "Time elapsed (row): "
         << diffclock(end,begin)
         << " ms" << endl;

    begin=clock();
    column_ordered();
    end=clock();
    cout << "Time elapsed (column): "
         << diffclock(end,begin)
         << " ms" << endl;

    return 0;
}
```