CPSC 131 – Data Structures

Analysis of Algorithms

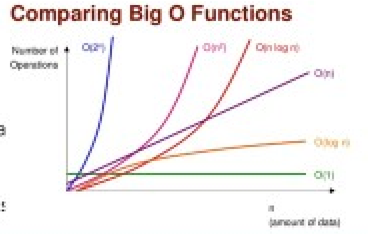
Professor T. L. Bettens
Spring 2023

Key terms

- **Asymptotic analysis**
- **Asymptotic efficiency class**
- **Worst-case analysis**
- **Big-Oh notation**
- **Constant time operations O(1)**
- **Logarithmic time operations $O(\log_2 n)$**
- **Linear time operations O(n)**
- **Quadratic time operations $O(n^2)$**

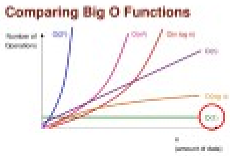
Common classes of Big-Oh functions

- O(1): Constant time operations
  – Does not depend on n
- O(n) : Linear time operations
  – Proportional to n
- O(log n) Logarithmic time opera
  – Bounded by $\log_2$ of n
- $O(n^2)$ Quadratic time operations
  – Unbounded



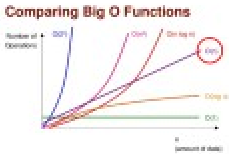
Efficiency Class Examples

- O(1): Constant time operations
- O(n) : Linear time operations
- O(log n): Logarithmic time operations
- $O(n^2)$: Quadratic time operations



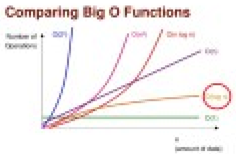
Efficiency Class Examples

- O(1): Constant time operations
- O(n) : Linear time operations
- O(log n): Logarithmic time operations
- $O(n^2)$: Quadratic time operations



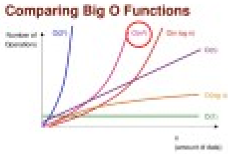
Efficiency Class Examples

- O(1): Constant time operations
- O(n) : Linear time operations
- O(log n): Logarithmic time operations
- $O(n^2)$: Quadratic time operations



Efficiency Class Examples

- O(1): Constant time operations
- O(n) : Linear time operations
- O(log n): Logarithmic time operations
- $O(n^2)$: Quadratic time operations



What about memory requirement?

- So far, we have been speaking of running time
- What about how much memory/space is occupied by a data structure?
- Can also use O(n) concept:
  – Does memory usage go up proportionately to number of elements?

CPSC 131 – Data Structures

Analysis of Algorithms

Professor T. L. Bettens

Spring 2023

Key terms

- **Asymptotic analysis**
- **Asymptotic efficiency class**
- **Worst-case analysis**
- **Big-Oh notation**
- **Constant time operations O(1)**
- **Logarithmic time operations $O(\log_2 n)$**
- **Linear time operations O(n)**
- **Quadratic time operations $O(n^2)$**

Comparing data structures

- Is a doubly linked list better than a singly linked list?

- What does it mean to be better? Under what conditions? What does "good" mean?
  - Running time? Memory used?
  - Usually means a trade-off

- Two approaches to answering this question:
  - Option 1: Experimental analysis
  - Option 2: Asymptotic analysis

Comparing

Data Structure

- Arrays
- Singly linked lists
- Doubly linked lists
- Binary Trees
- Hash Tables

Operations

- Create empty
- Get front element
- Add or remove front element
- Get back element
- Add or remove back element
- Clear data structure
- Get/add/remove i^{th} element

Comparing

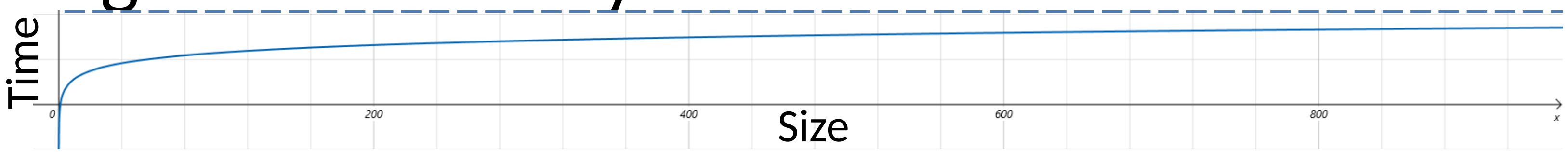
- This course's goal: the design of "good" data structures and algorithms
- Data structures: systematic way of organizing and accessing data
- Algorithms: Step-by-step procedure for performing a task in a finite time.
- What does "good" mean?
  - Running time—fast
  - Space usage—small
- Usually means a trade-off
  - Faster often requires more memory for extra pointers
  - Smaller often requires more complex algorithms
- Essential point: **Running time increases with input size**

Option 2: Asymptotic Analysis

- Analysis without actually running any code
  - Done at the desktop looking at source code

Look for the loops!

- Asymptotic: approaching a value closely



- Key idea:
  - We are interested in running time for large data sets
  - How fast will running time increase as we increase data size?
  - Rate of increase

Option 2: Asymptotic Analysis

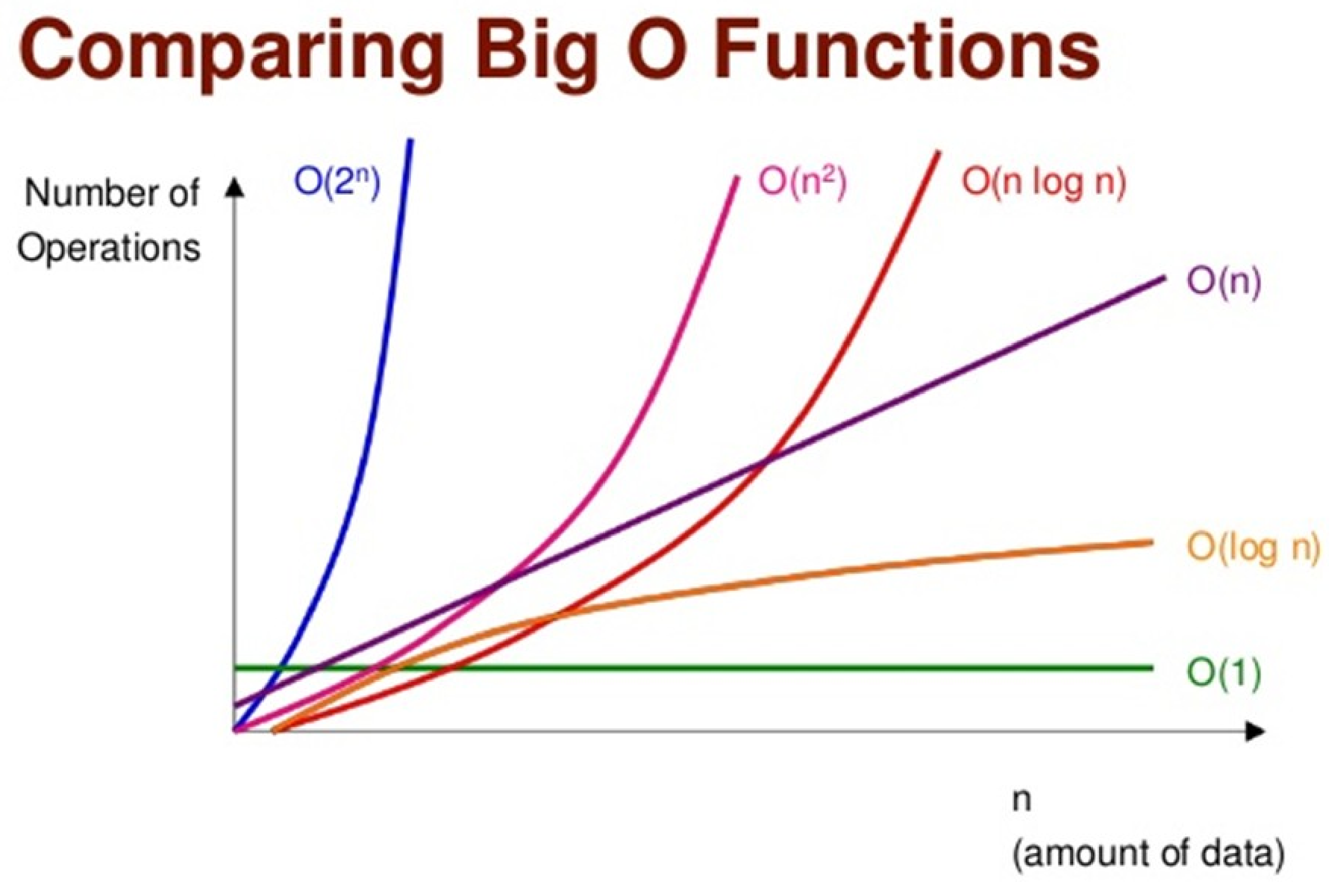
- Most important factor?
  - Number of elements in the data structure (i.e., size)

- Represent this by ***n***

- Analysis
  - How does running time increase in terms of n?

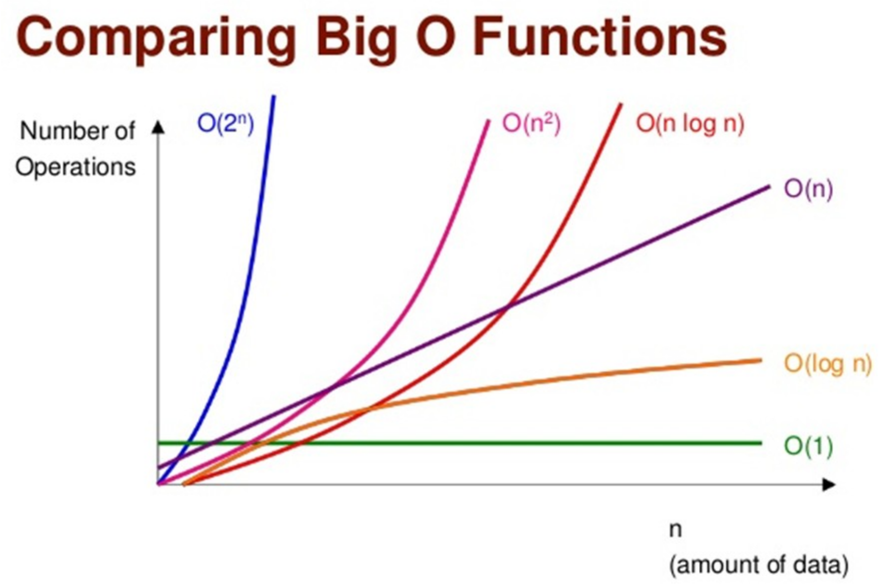
Common classes of Big-Oh functions

- O(1): Constant time operations
  - Does not depend on n
- O(n) : Linear time operations
  - Proportional to n
- O(log n) Logarithmic time operatio
  - Bounded by $\log_2$ of n
- $O(n^2)$ Quadratic time operations
  - Unbounded



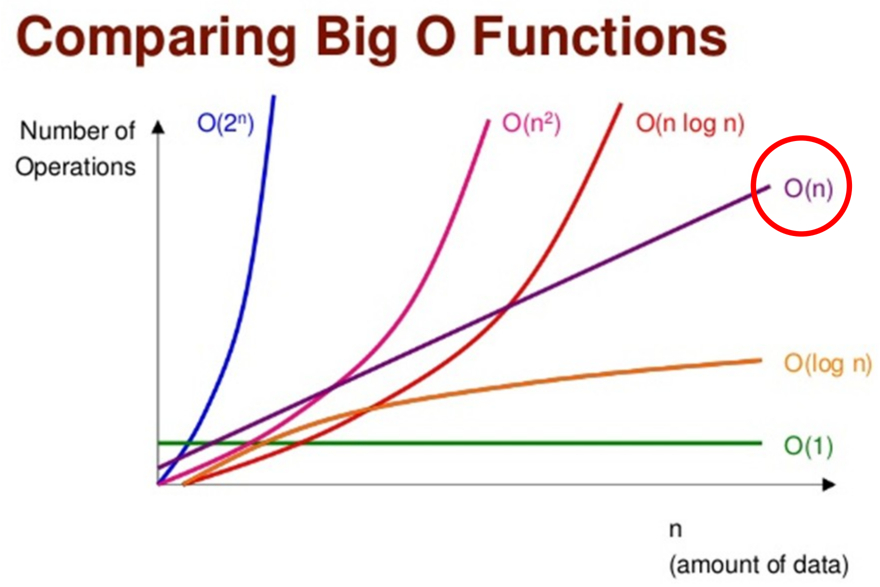
Efficiency Class Examples

- O(1): Constant time operations
- O(n) : Linear time operations
- O(log n): Logarithmic time operations
- O(n^2): Quadratic time operations



Efficiency Class Examples

- O(1): Constant time operations
- O(n) : Linear time operations
- O(log n): Logarithmic time operations
- O(n^2): Quadratic time operations



Efficiency Class Example: O(n)
Printing a list

- Consider a list of n elements

```
for (const auto & value : list )
{
  cout << value << endl;
}
```

Look for the loops!

- How many steps did we have to do?
  - Too complex!
- Do the number of steps increase in proportion to **n**?

Efficiency Class Example: O(n)
Printing a list

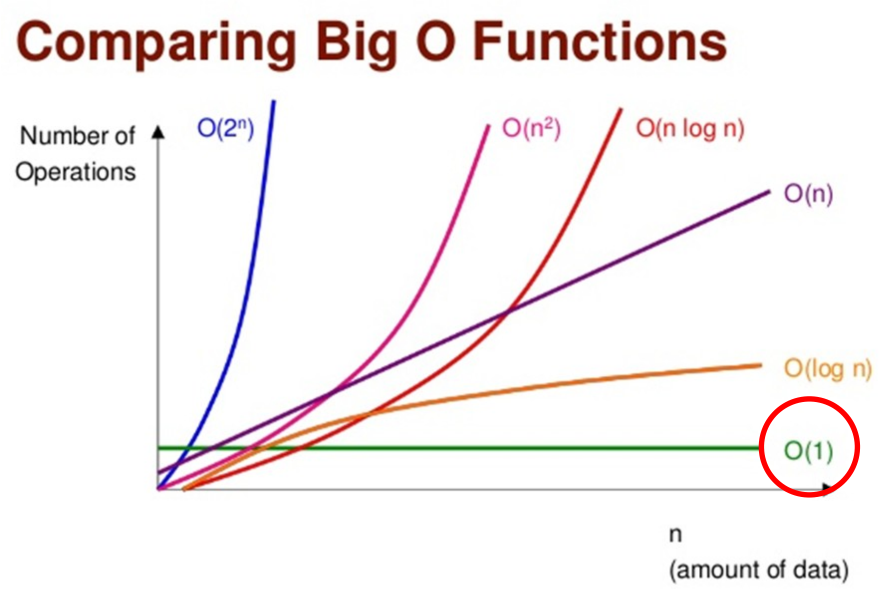
- Yes! number of operations increase in proportion to n
- "Printing all elements in a list takes on the order of n"
- Written as O(n)
- Also commonly spoken as "Oh of n"

Efficiency Class Example: O(n)
Printing a list

- But what about the fact that we had to
  - initialize the loop
  - Printing required cout
  - We also printed endl
  - …
- Don't care about constant factors
  - Focus on the big picture
    - Not details like initialization
- Worst-case analysis

Efficiency Class Examples

- O(1): Constant time operations
- O(n) : Linear time operations
- O(log n): Logarithmic time operations
- O(n^2): Quadratic time operations



Efficiency Class Example: O(1)
Algorithm Independent of "n"

- Return the value of an attribute (e.g., getters)

```
std::size_t size() { return _size; }
```

- Fixed length loops

```
for (unsigned int i = 0; i < 10; ++i )
{
  cout << i << '\n';
}
```

- Math expressions

```
limit = 2 * sin(x+y) + 3 * z;
```

Look for the loops!

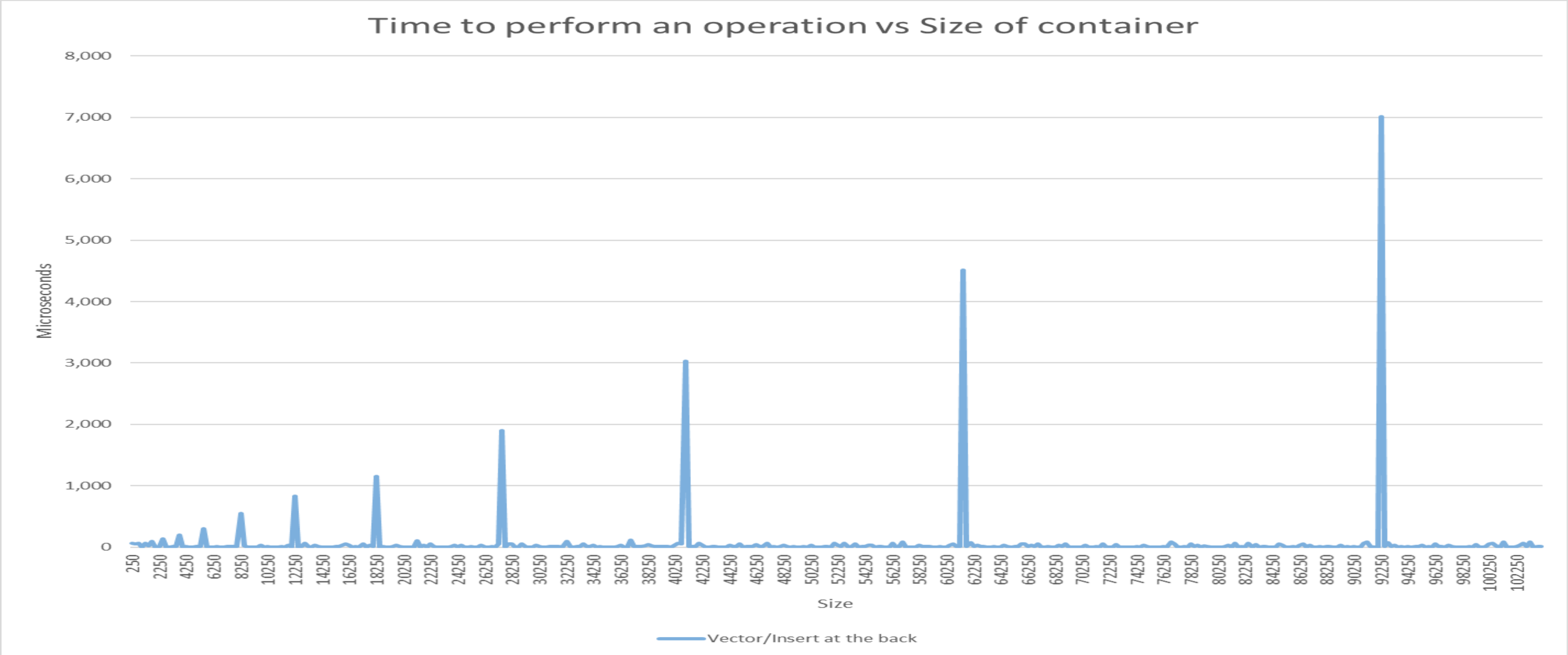
But make sure the loops are dependent on 'n'

Efficiency Class Example: O(1) Amortization

- Financial: Amortization is paying off an amount owed over time by making planned, incremental payments of principal and interest.

- Computer Science: To even out the costs of running an algorithm over many iterations, so that high-cost iterations are much less frequent than low-cost iterations, which lowers the average running time per iteration.

Efficiency Class Example: Amortized O(1)



Source: ref2

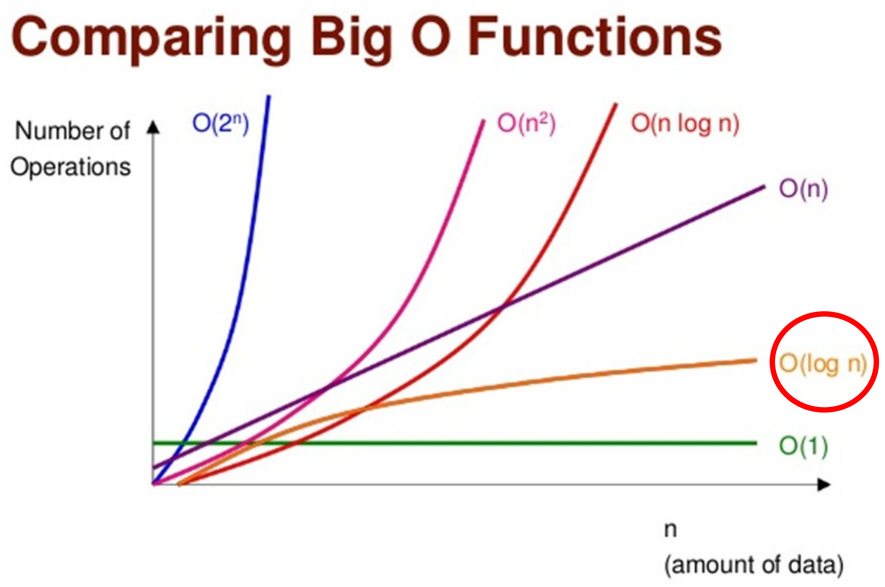
Efficiency Class Example: Amortized O(1)

Increase the Extension Size, Reduce the Number of Copy Operations

	Extend by 1			Double Each Extension		
Insert #	Size	Capacity	Copies	Size	Capacity	Copies
1	1	1	0	1	1	
2	2	2	1	2	2	1
3	3	3	2	3	4	2
4	4	4	3	4	4	
5	5	5	4	5	8	4
6	6	6	5	6	8	
7	7	7	6	7	8	
8	8	8	7	8	8	
9	9	9	8	9	16	8
10	10	10	9	10	16	
11	11	11	10	11	16	
12	12	12	11	12	16	
13	13	13	12	13	16	
14	14	14	13	14	16	
15	15	15	14	15	16	
16	16	16	15	16	16	
		Total Copies	**120**		Total Copies	**15**

Efficiency Class Examples

- $O(1)$: Constant time operations
- $O(n)$: Linear time operations
- $O(\log n)$: Logarithmic time operations
- $O(n^2)$: Quadratic time operations



Source: ref2

Efficiency Class Example: O(log n)
Binary Search

Search for 75

Round 1

0	1	2	3	4	5	6	7	8
10	20	30	40	50	60	70	80	90
lwr				mid				upr

Round 2

0	1	2	3	4	5	6	7	8
10	20	30	40	50	60	70	80	90
					lwr	mid		upr

Round 3

0	1	2	3	4	5	6	7	8
10	20	30	40	50	60	70	80	90
							lwr mid	upr

Round 4

0	1	2	3	4	5	6	7	8
10	20	30	40	50	60	70	80	90
						upr	lwr	

range is empty

N = 9

75 == 50? (1)

75 == 70? (2)

75 == 80? (3)

(4)

- $\log_2(9) = 3.17 <= 4$
- Any n between 9 and 16 will take at most 4 operations
- Binary search of sorted data: O(log n)

$Log_2(n)$ Example: Binary Search

Search for 75

Round 1

0	1	2	3	4	5	6	7	8
10	20	30	40	50	60	70	80	90
lwr				mid				upr

Round 2

0	1	2	3	4	5	6	7	8
10	20	30	40	50	60	70	80	90
					lwr	mid		upr

Round 3

0	1	2	3	4	5	6	7	8
10	20	30	40	50	60	70	80	90
							lwr mid	upr

Round 4

0	1	2	3	4	5	6	7	8
10	20	30	40	50	60	70	80	90
						upr	lwr	

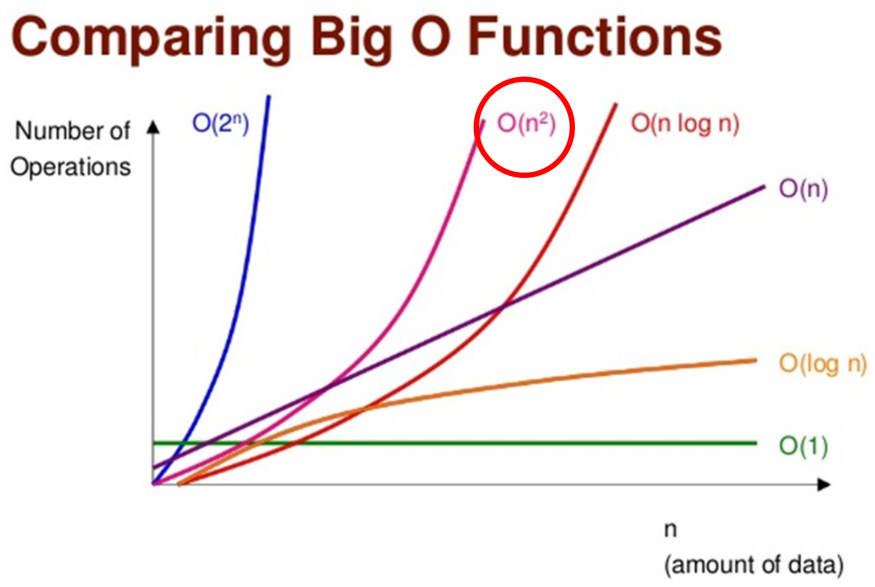
range is empty

```
size_t search( const array<int, 9> & c, int
v )
{
  size_t s       = c.size();
  size_t current = s / 2;

  while( s != 0 )
  {
    if( v == c[current] ) return current;
    if( s == 1          ) break;

    s = ( s + 1 ) / 2;

    if( v < c[current] ) current -= s / 2;
    else                 current += s / 2;
  }

  return numeric_limits<size_t>::max();
}
```

Look for the loops!

Efficiency Class Examples

- O(1): Constant time operations
- O(n) : Linear time operations
- O(log n): Logarithmic time operations
- O(n^2): Quadratic time operations



Efficiency Class Example: $O(n^2)$
Visit each cell of a Matrix

N = 4

4 X 4 Matrix

For each row r visit each column c

	0	1	2	3
0	(0, 0)	(0, 1)	(0, 2)	(0, 3)
1	(1, 0)	(1, 1)	(1, 2)	(1, 3)
2	(2, 0)	(2, 1)	(2, 2)	(2, 3)
3	(3, 0)	(3, 1)	(3, 2)	(3, 3)

Efficiency Class Example: $O(n^2)$
Visit each cell of a Matrix

```
// Defines an NxN matrix containing elements of type E
template<typename E, std::size_t N>
using Matrix = std::array<std::array<E, N>, N>;

// Determine if an NxN matrix is symmetrical along its major axis
template<typename E, std::size_t N>
bool isSymmetrical( const Matrix<E, N> & matrix )
{
  for( std::size_t row = 0; row < matrix     .size(); ++row )
  for( std::size_t col = 0; col < matrix[row].size(); ++col )
  {
    if( ( row != col ) && ( matrix[row][col] != matrix[col][row] ) ) return false;
  }

  return true;
}
```

Look for the loops!

In this case, Nested loops!

What about memory requirement?

- So far, we have been speaking of running time
- What about how much memory/space is occupied by a data structure?
- Can also use O(n) concept:
  - Does memory usage go up proportionately to number of elements?