---

complexity analysis

- **Definition:**

  Complexity analysis categorizes an algorithm based on resource usage in relation to the input size (n).

- **Time Complexity:**

  A categorization related to time.

- **Space Complexity:**

  A categorization related to space.

- **Example:**

  To sort all of the names of every person in the world, we can choose either an n^2 or nlgn algorithm

  There are 6.975 billion people in the world (n). The time to execute depends on the time complexity.

  Assume we have to do 1 nanosecond of computation per sort operation.

  - **O(n^2) bubblesort:**
    - n^2 = 6.975 billion ^ 2 = 4.8509e19 operations
    - seconds = 4.8509e19 operations * 1 (nanosecond / operation)
    - seconds = 4.86e10
    - years = 4.86e10 seconds / 31,536,000 (seconds / year)
    - years = 1,541
  - **O(nlgn) heapsort:**
    - nlgn = 6.975 billion * lg(6.975 billion) = 2.3001e11 operations
    - seconds = 2.3e11 operations * 1 (nanosecond / operation)
    - seconds = 230
    - minutes = 230 seconds / 60 (seconds / minute)
    - minutes = 3.8

  You can program a bubblesort in 5 minutes or you can program a heapsort in 1 hour. The purpose of complexity analysis is to do back of the envelope calculations to gauge how much programming time should be used optimizing performance. Or to see if an operation on a data input size is feasible within our lifetime with our most advanced algorithms.

  Usually we want to choose an algorithm with a better time or space complexity

  - But this is not always true!
    - O(n^2) is really an^2 + bn + c
    - O(nlgn) is really anlgn + bn + c
    - Sometimes the constant factors of an O(n^2) algorithm are much smaller than, say, an O(nlgn) algorithm for small n
  - Sometimes we have to choose either a good time complexity or a good space complexity

---

growth of functions

- f(n) = n^2 + 50nlgn + 100n + lgn + 1000

n	f(n)	n^2		50nlgn		100n		lgn		1000	
		Value	%	Value	%	Value	%	Value	%	Value	%
2	1305	4	0.31	100	7.66	200	15.33	1	0.08	1000	76.63
16	6060	256	4.22	3200	52.81	1600	26.40	4	0.07	1000	16.50
256	194544	65536	33.69	102400	52.64	25600	13.16	8	0.00	1000	0.51
4096	19645428	16777216	85.40	2457600	12.51	409600	2.08	12	0.00	1000	0.01
65536	4353950712	4294967296	98.65	52428800	1.20	6553600	0.15	16	0.00	1000	0.00

---

common cyclomatic complexities

- O(n^3)
- O(n^2)
- O(nlgn)
- O(n)
- O(lgn)
- O(1)

Draw graph of this function including the constant term on the board

---

lgn

- We use log base 2 in computer science often
- lgn is the number of times you divide n by 2 until you get to one
- Often lgn comes up as part of a complexity because we divide a problem into two until we cannot anymore

An example log base 2 calculator:

```
// logBase2.cpp - download here

#include <iostream>

int logBase2(int n){
  int ret = 0;
  while(n > 1){
    n /= 2;
    ret++;
  }
  return ret;
}

int main(int argc, char * argv[]){

  for(int i = 0; i <= 32; ++i){
    int log2 = logBase2(i);
    std::cout << "the log base 2 of (" << i << ") = " << log2 << std::endl;
  }

  return 0;
```

```
}
```

The results:

- lg(0) = 0
- lg(1) = 0
- lg(2) = 1
- lg(4) = 2
- lg(8) = 3
- lg(16) = 4
- lg(32) = 5
- lg(64) = 6
- lg(128) = 7
- lg(256) = 8
- lg(512) = 9
- lg(1024) = 10
- lg(4*1024*1024) = 32

binary for 256: 0b100000000.
a single high bit and 8 zero in the binary number.
lg(256) = 8.
each leading one creates a new lgn increment because binary is base2.

lgn grows extremely slowly compared to n. so nlgn is much better than n^2.

---

big-o notation

- f(n) = n^2 + 50nlgn + 100n + lgn + 1000
- O(f(n)) = O(n^2)
- As n becomes large, the terms that grow smaller than n^2 do not matter
- Big O defines the upper bounds of functions. In the worst possible performing case of an algorithm, it will never be greater than O(f(n)) by more than a constant factor.

- Big O is a best possible worst case.
- f(n) = 3n^2
  - O(f(n)) = O(n^2)
  - O(f(n)) != O(n^3)
  - O(f(n)) != O(n)

---

complexity class examples

- O(1) - Simple Calculation: Conversion from Fahrenheit to Celsius

```
// temperature.cpp - download here

float toCelsius(float fahrenheit_temp){
  float ret = (fahrenheit_temp - 32) * 5.0 / 9.0;
  return ret;
}
```

- O(lgn) - Binary Search: Given a sorted sequence, find if an element exists in the sequence

```
// binarySearch.cpp - download here

int binarySearch(int * array, int n, int key){
  int low = 0;
  int mid;
  int high = n-1;

  while(low <= high){
    mid = (low + high) / 2;
    if(key < array[mid]){
      high = mid - 1;
    } else if(array[mid] < key){
      low = mid + 1;
    } else {
      return mid;
    }
  }
  return -1;
}
```

- O(n) - Compute the Sum of an Array

```
// computeSum.cpp - download here

int computeSum(int * array, int n){
  int ret = 0;
  for(int i = 0; i < n; ++i){
    ret += array[i];
  }
  return ret;
}
```

- O(n) - Moving Average: Just because there are two for loops doesn't mean it is O(n^2)

```
// movingAverage.cpp - download here

int * movingAverage(int * data, int n, int window_size){
  int * ret = new int[n];
  int half_window = window_size / 2;
  for(int i = 0; i < n; ++i){
    int sum = 0;
    int num_counted = 0;
    for(int j = i - half_window; j < i + half_window; ++j){
      if(j >= 0 && j < n){
        num_counted++;
        sum += data[j];
      }
      ret[i] = sum / num_counted;
    }
  }
  return ret;
}
```

- O(nlgn) - Efficient Sort (Merge Sort), We'll study this later, don't worry about it for now.

```
// mergeSort.cpp - download here

#include <iostream>

int * merge(int * left, int * right, int len){
  int * ret = new int[len + len];
```

```
  int left_position = 0;
  int right_position = 0;
  int ret_position = 0;
  while(left_position < len && right_position < len){
    int left_value = left[left_position];
    int right_value = right[right_position];
    if(left_value < right_value){
      ret[ret_position] = left_value;
      ret_position++;
      left_position++;
    } else {
      ret[ret_position] = right_value;
      ret_position++;
      right_position++;
    }
  }
  while(left_position < len){
    ret[ret_position] = left[left_position];
    ret_position++;
    left_position++;
  }
  while(right_position < len){
    ret[ret_position] = right[right_position];
    ret_position++;
    right_position++;
  }
  return ret;
}

int * mergeSort(int * input, int len){
  if(len == 1){
    return input;
  }
  int middle = len / 2;
  int * left = new int[middle];
  int * right = new int[middle];
  for(int i = 0; i < middle; ++i){
    left[i] = input[i];
  }
  for(int i = 0; i < middle; ++i){
    right[i] = input[i+middle];
  }
  left = mergeSort(left, middle);
  right = mergeSort(right, middle);
  int * ret = merge(left, right, middle);
  delete [] left;
  delete [] right;
  return ret;
}

int main(int argc, char * argv[]){

  int * array = new int[8];
  array[0] = 8;
  array[1] = 2;
  array[2] = 4;
  array[3] = 9;
  array[4] = 3;
  array[5] = 6;
  array[6] = 10;
  array[7] = 5;

  int * sorted = mergeSort(array, 8);
  for(int i = 0; i < 8; ++i){
    std::cout << sorted[i] << " ";
  }
  std::cout << std::endl;

```

```
  return 0;
}
```

- O(n^2) **-** Bubble Sort: given any sequence, ensure all the elements are in ascending (or descending) order

```
// bubbleSort.cpp - download here

void bubbleSort(int * array, int n){
  for(int i = 0; i < n; ++i){
    for(int j = i + 1; j < n; ++j){
      int lhs = array[i];
      int rhs = array[j];
      if(rhs < lhs){
        array[i] = rhs;
        array[j] = lhs;
      }
    }
  }
}
```

---

references

1. http://en.wikipedia.org/wiki/Big_O_notation
2. http://en.wikipedia.org/wiki/World_population
3. http://www.cs.rpi.edu/academics/courses/fall11/ds/lectures/01_intro.pdf
4. Adam Drozdek. "Data Structures and Algorithms in C++"
