Data Structures in C

Prof. Georg Feil

Computational Complexity and Big-O Notation

Winter 2018

Acknowledgement

- These lecture slides are partly based on slides by Professor Simon Hood
- Additional sources are cited separately

Reading Assignment (required)

- Read **A beginner's guide to Big O notation**
- Read Data Structures (recommended textbook)
  - Chapter 1 sections 1.1, 1.6, 1.7

Note the textbook does a few things we might consider poor style, for example one-letter variable names and using int for Boolean values.

Computational Complexity

- When we say “computational complexity” we don’t mean an algorithm is complex or complicated
- The complexity of an algorithm refers to how efficient or fast it is for different amounts of input data
  - e.g. Look up a person’s information in country’s tax database.
    - What if the country has 100,000 taxpayers? 1,000,000? 99,000,000?
- We can analyze algorithms to understand how they behave as the amount of data processed grows
- Many different algorithms have essentially equivalent complexity even though their code is quite different
  - They belong to the same complexity class

Big O notation

- It's not enough to just write code (many people can do that) – we want code to run as quickly and efficiently as possible
- The "O" stands for order
- We can use big O notation to describe and compare the efficiency of algorithms
  - Two algorithms with the same order (big O) are in the same complexity class and their efficiency is about the same
  - e.g. The efficiency of selection sort and insertion sort is similar
- With big O we're concerned about the worst case of an algorithm
  - If it runs really fast for some inputs that's not important

Common Big O values (complexity classes)

O(1)

- *Constant* time
- This describes an algorithm that always takes about the same time regardless of the size of input data
- Example: Accessing a particular element of an array

```
result = arr[10];
```

- This takes about the same amount of time regardless of the size of the array or which element you access

Common Big O values (complexity classes)

O(n)

- *Linear* time
- The time taken grows linearly, so if we double the size of input data, the algorithm takes twice as long
- Example one: A simple loop

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

Common Big O values (complexity classes)

O(n)

- Example two: Linear search for ‘value’

```
for (int i = 0; i < length; i++) {
    if (data[i] == value)
        return true;  // Found
}
return false;  // Not found
```

- Sometimes this might find the item we’re looking for right away, but remember we’re concerned about the worst case so it’s O(n)   *(what is ‘n’ in this example?)*

Common Big O values (complexity classes)

$O(n^2)$

- *Quadratic* time
- The time taken is proportional to the square of the input data size
  - If we double the size of input data, the algorithm takes 4 x as long
- Example: Bubble sort

```
for (int i = 0; i < n; i++) {
    for (int j = 0; j < n; j++) {
        // compare values and swap if necessary
    }
}
```

- When you see a nested loop the algorithm is probably $O(n^2)$

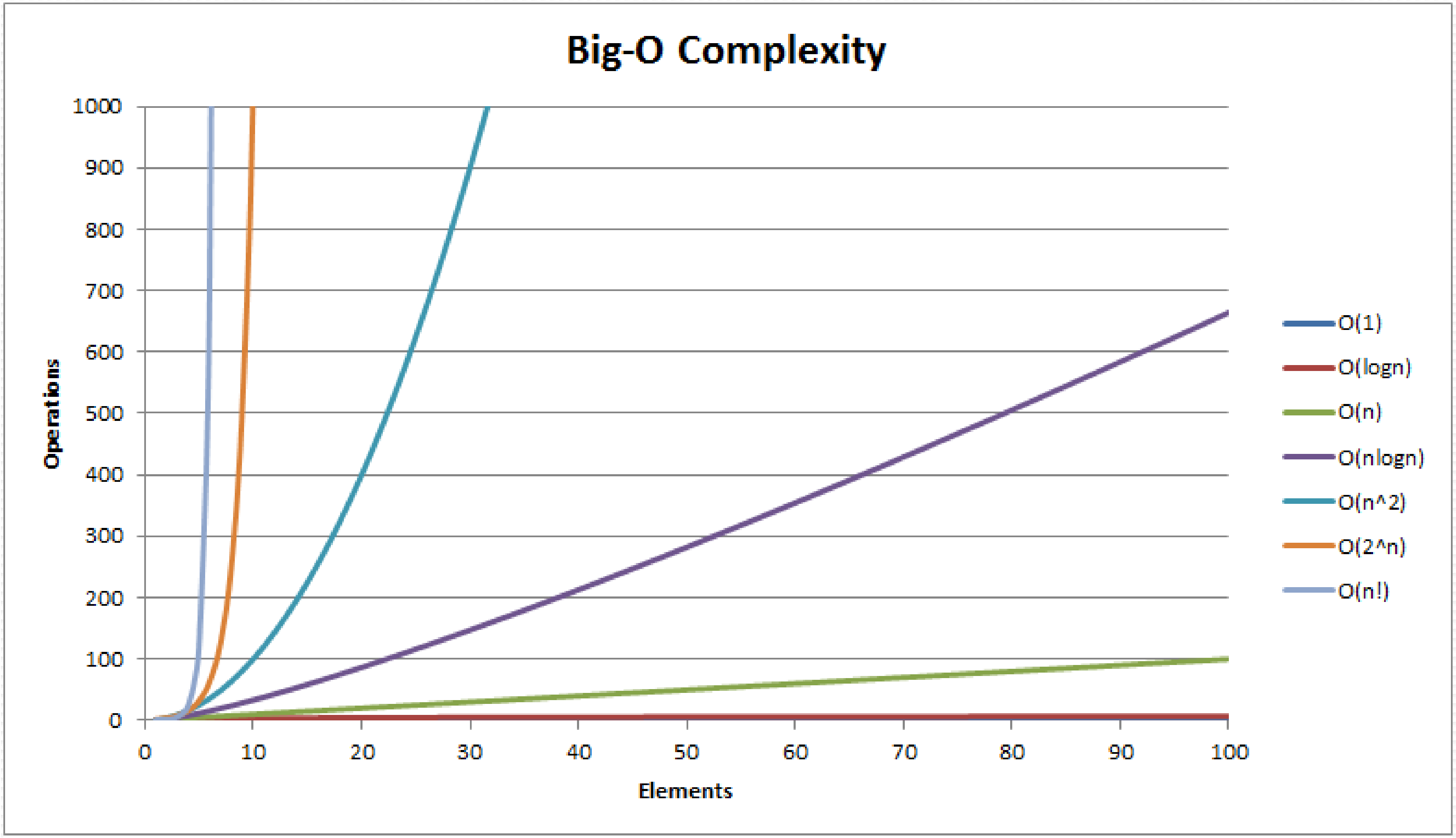
Common Big O values (complexity classes)

$O(n^2)$

- How many times does the nested loop go around?
- For every value up to n (outer loop) we do n iterations (inner loop)

  → $n * n$ or n^2

Common Big O values (complexity classes)



Common Big O values (complexity classes)

- Wait... we can make bubble sort more efficient like this

```
for (int i = 0; i < n; i++) {
    for (int j = 0; j < i; j++) {
        // compare values and swap if necessary
    }
}
```

- In this improved bubble sort the inner loop will iterate only ‘i’ times. Isn’t this n * i?
- For Big O though, we deal in terms of worst-case complexity, and the worst case here is i == n, so n * n
- It is still, therefore, $O(n^2)$

Constants

- We also drop constants when determining Big O

```
// linear?
for (int i= 0; i < 2*n; i++) {
    sum = sum + i;
}
```

- This appears to be of complexity O(2n)
- However, we drop constants, so it ends up being O(n)
- Mathematically, it requires on the order of ‘n’ iterations even though we literally iterate 2n times

Adding complexities

- What about sequential loops? Consider the following code

```
// linear
for (int i = 0; i < n; i++) { … }
// quadratic
for (int i = 0; i < n; i++) {
    for (int j = 0; j < n; j++) { … }
}
```

Adding complexities

- To determine Big O, we add each loop's Big O together
  - In this case, it's $n + n^2$
- But as 'n' gets very big (limit as 'n' approaches infinity), n^2 will dwarf the n term!
- We almost always drop the smaller terms when adding
- That means the previous code is still $O(n^2)$!

Loops that don’t grow

- Loops with specific endpoints are common in programming

```
// quadratic?
for (int i = 0; i< n; i++) {
    for (int j = 0; j < 8; j++) { … }
}
```

- Outer loop is n, inner loop is 8, so we have n * 8
- We drop constants though, so this example is still linear with a time of O(n)

More advanced algorithm complexities

(But still common)

Advanced Big O values

O(log n)

- *Logarithmic* time
- Examine this loop:

```
for (int i = 1; i < n; i=i*2) {
    sum = sum + i;
}
```

- This loop doesn’t run ‘n’ times
  - it’s much faster than that…

Advanced Big O values

O(log n)

- The i=i*2 operation means this loop doesn't run n times
  - The larger 'i' gets, the faster it approaches the loop endpoint
  - 'i' grows exponentially, making the run time grow logarithmically
- If n = 10,
  - The linear loop iterates 1 2 3 4 5 6 7 8 9 10
  - The log n loop iterates 1 2 4 8
- If n = 100,
  - The linear loop iterates 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32 33 34 35 36 37 38 39 40 41 42 43 44 45 46 47 48 49 50 51 52 53 54 55 56 57 58 59 60 61 62 63 64 65 66 66 68 69 70 71 72 73 74 75 76 77 78 79 80 81 82 83 84 … 100
  - The log n loop iterates 1 2 4 8 16 32 64

Advanced Big O values

O(log n)

- Log time is considered very fast for most algorithms
- Example: Binary search (e.g. looking up a word in a dictionary) is O(log n)
  - Try it with n = 1 million…

Advanced Big O values

O(n log n)

- We'll see many sorting algorithms that run in O(n log n) time

```
// n log n
for (int i = 0; i < n; i++) {
    for (int j = 1; j < n; j *= 2) { … }
}
```

- The outer loop runs n times, the inner runs log n times, so the Big O is n * log n or O(n log n)
- This doesn't seem good at first, but it's much better than $O(n^2)$

Exponential complexity

- Algorithms can have exponential complexity
- Example: Count the number of combinations of a series of numbers
- In this case we have something like $O(10^n)$ or $O(a^n)$
- This is considered unbelievably bad!!
  - It's known as exponential time or EXP, and grows very quickly for large values of n
- Important: Don't confuse this with polynomial complexities like $O(n^2)$, $O(n^3)$, $O(n^4)$ etc.
  - Notice in each of these the exponent is constant

Execution times for different complexities

Input Size (n)	Time Complexity				
	n	$n \log_2 n$	n^2	n^3	2^n
10	$< .001$ second	$< .001$ second	$< .001$ second	$< .001$ second	$< .001$ second
20	$< .001$ second	$< .001$ second	$< .001$ second	$< .001$ second	.001 second
30	$< .001$ second	$< .001$ second	$< .001$ second	$< .001$ second	1 second
50	$< .001$ second	$< .001$ second	$< .001$ second	$< .001$ second	13 days
100	$< .001$ second	$< .001$ second	$< .001$ second	.001 second	4×10^{11} centuries
1000	$< .001$ second	$< .001$ second	.001 second	1 second	4×10^{282} centuries
100,000	$< .001$ second	.002 second	10 seconds	11.57 days	–
one million	.001 second	.02 second	1.67 minutes	32 years	–
ten million	.01 second	0.24 second	1.2 days	317 centuries	–
one billion	1 second	30 seconds	32 years	4×10^8 centuries	–
100 billion	1.67 minutes	1 hour	3171 centuries	4×10^{14} centuries	–

“Hard” complexities

- There is a whole field of computer science theory behind something called P vs. NP and NP-complete problems
  - P = polynomial time
  - NP = non-deterministic polynomial time
- Nobody has been able to prove whether NP-complete problems can be solved in polynomial time
  - It’s one of the great unsolved mysteries of computer science
  - Algorithms which solve NP-complete problems are exponential
- NP-complete problems are considered among the hardest to code efficiently, and interestingly, are all mathematically provable to be the same problem!

“Hard” complexities

O(n!)

- The worst complexity I’ll mention today is O(n!), or factorial time
  - It’s worse than exponential
- A true O(n!) problem is the travelling salesman problem
  - Note that this problem is also NP-complete

Travelling Salesman problem – O(n!)

- A salesman wants to visit a number of cities, but he wants to visit them using the least amount of gas – can we find a route among all the cities that follows the shortest path?
  - There are n possible paths for the first city. For the next choice, there are n-1 paths to choose. For the city after that, there are n-2 paths, etc.
  - n * (n-1) * (n-2) … = n!
- If we consider just 15 cities, there are 1,307,674,368,000 path choices
- This means that this problem will almost never be solved for any reasonable value of n using just *brute force*

Big O determination summary

- When determining the big O formula for an algorithm:

1. Nested loops are multiplied together
2. Sequential loops are added …
3. … but usually only the largest term is kept – all others are dropped
4. Constants are dropped
5. Conditional checks are constant

Exercises

- See “Big O Exercises located in the same folder as these slides in SLATE