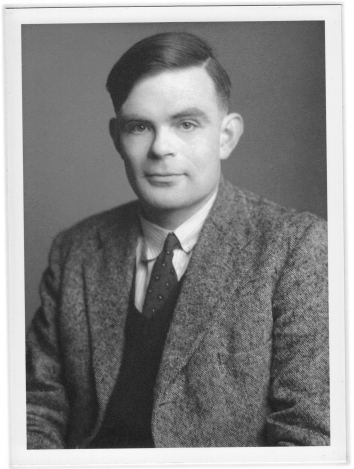
CompSci 201, L8: Asymptotic (Big-O) Analysis

Logistics, Coming up

- Today, 2/8
  - APT 3 due
- Next Monday 2/13
  - Midterm exam 1
- Next Wednesday 2/15
  - APT 4 due

Person in CS: Alan Turing

- 1912-1954 (died at 41)
- English, PhD at Princeton in 1938
- Mathematician, cryptographer, pioneering thinker in AI
  - "Father of modern computer science"
  - Turing machine – helped formalize what is computable
  - Cryptography work in WW2
- Prosecuted in 1952 for homosexuality
  - Given choice of chemical "treatment" or prison, took former
  - Died 2 years later of cyanide poisoning, circumstances debated

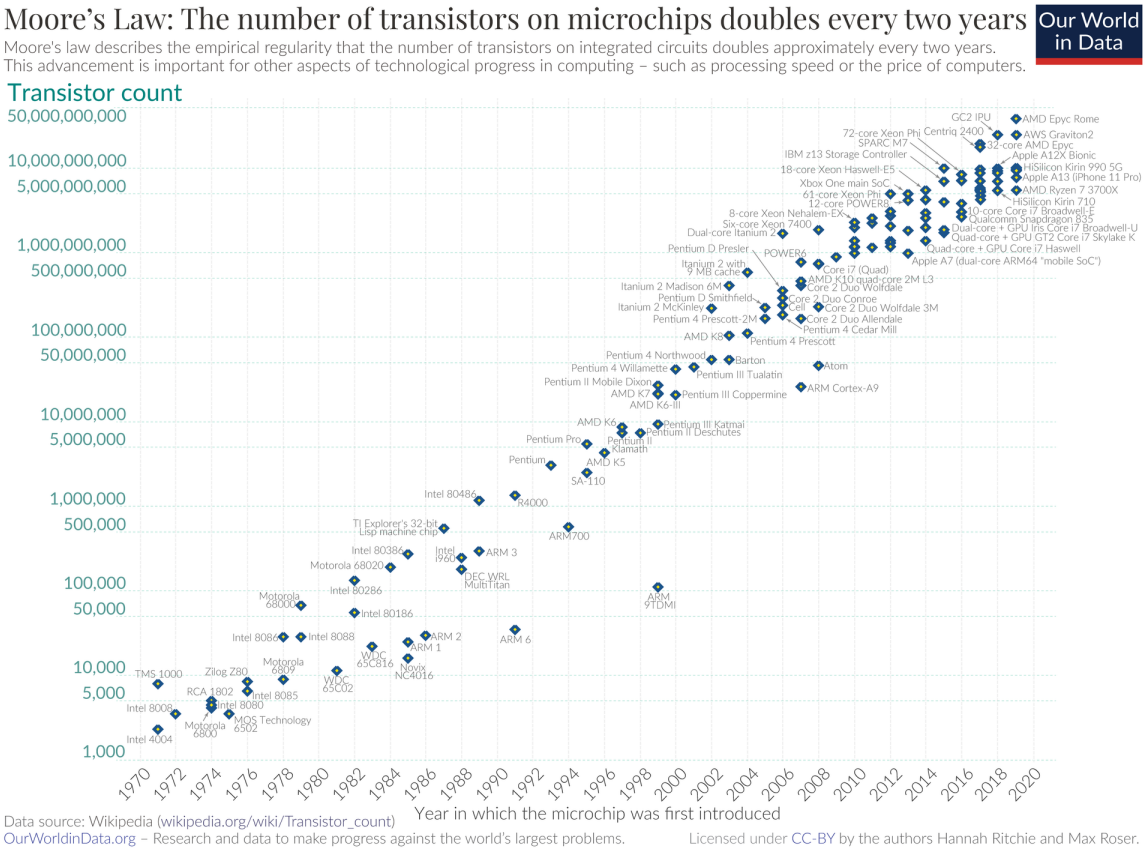


Asymptotic Analysis and Big O Notation

Runtime and memory

- Two most fundamental resources on a computer:
  - Processor cycles: Number of operations per second machine can perform
    - (2 GHz = 2 billion operations per second).
  - Memory: space for storing variables, data, etc.
    - (esp. working memory, a.k.a. cache and RAM)
- We will mostly focus on runtime complexity
  - Often comes at expense of memory, e.g., HashMap
- Start by reasoning about empirical runtimes, but…

Problem with empirical runtimes



Same code that takes 1 min. in 1990 takes

- ~2 s in 2000?
- ~63 ms in 2010?
- ~2 ms in 2020?

How do we measure efficiency of the code apart from the machine?

- Let N be the size of the input
  - For some `int[] ar`, N could be `ar.length`
- Count T(N) = number of *constant time* operations in the code as a function of N.
- Reason about how T(N) grows when N becomes large.
  - "Asymptotic" (in the limit) notation

Reminder: What is constant time?

- Running time *does not depend on size of the input*.
  - If ~1 ms to `.get()` when ArrayList has 1,000 elements?
  - Then ~1 ms to `.get()` when ArrayList has 1,000,000 values.

- Other constant time operations might be a *very different* constant.
  - Adding 2+2 might be faster than .get(), but both are constant.

Constant Time Examples

- Index into an array (ar[0] or ar[201])
- Arithmetic (`+`, `-`, `*`, `/`, `%`, etc.)
- Primitive comparison `<`, `==`, etc.
- Access an object attribute (e.g. `.length`)
- ArrayList `.get()`, `.size()`, `.add()` [to end, amortized]

- Non-constant time usually has a loop or method call, may depend on data structure implementation

Big-O (limit definition)

- Given N (for example, the size of the input)
- Function T(N) (for example, the number of *constant time* operations in the code)

Definition (big *O notation*). $T(N)$ is $O(g(N))$ if $\lim_{N\to\infty} \frac{T(N)}{g(N)} \leq c$ for some constant c that does not depend on N.

In other words: $T(N)$ is $O(g(N))$ if it is at most a constant factor times slower than $g(N)$ for large input N.

Two general rules

1. Can drop constants
   - $2N+3 \rightarrow O(N)$
   - $0.001N + 1{,}000{,}000 \rightarrow O(N)$

2. Can drop lower order terms
   - $2N^2+3N \rightarrow O(N^2)$
   - $N+\log(N) \rightarrow O(N)$
   - $2^N + N^2 \rightarrow O(2^N)$

Hiearchy of some common complexity class

Big O	Name	Example
$O(2^N)$	Exponential	Calculate all subsets of a set
$O(N^3)$	Cubic	Multiply NxN matrices
$O(N^2)$	Quadratic	Loop over all *pairs* from N things
O(N log(N))	Nearly-linear	Sorting algorithms
O(N)	Linear	Loop over N things
O(log(N))	Logarithmic	Binary search a sorted list
O(1)	Constant	Addition, array access, etc.

Some common complexity classes and their growth

N	O(log(N))	O(N)	$O(N^2)$	$O(N^3)$	$O(2^N)$
1	1	1	1	1	2
2	2	2	4	8	4
4	3	4	16	64	16
8	4	8	64	512	256
16	5	16	256	4k	65k
32	6	32	1k	32k	4.2E+9
64	7	64	4k	262k	1.8E+19

If you double N…

- O(log(N)) adds ~1
- O(N) roughly doubles
- $O(N^2)$ roughly quadruples
- $O(N^3)$ roughly multiples by 8
- $O(2^N)$ squares each time

Note: Turning these into runtimes depends on your machine.

Relation to Empirical Timing and Lower Order Terms

N	n^2 + 19n + 200	factor increase
10	490	NA
20	980	**2.00**
40	2560	2.61
80	8120	3.17
160	28840	3.55
320	108680	3.77
640	421960	3.88
1280	1662920	3.94
2560	6602440	3.97
5120	26311880	3.99
10240	105052360	3.99
20480	419819720	**4.00**

Looks linear?

Asymptotic analysis describe behavior *in the limit as n becomes large*, lower order terms may dominate at small input sizes.

Looks quadratic?

WOTO

Go to duke.is/6ezc8

Not graded for correctness, just participation.

Try to answer *without* looking back at slides and notes.

But do talk to your neighbors!

L08-WOTO1-BigO

* Required

* This form will record your name, please fill your name.

1

NetID *

2

n^2 + nlog(n) + (log(n))^2 is... *

- O(n^2)
- O(nlog(n))
- O((log(n))^2)

3

n^2 + 2^n is... *

- ◯ O(n^2)
- ◯ O(2^n)

4

log(n^2) is... *

- ◯ O(n^2)
- ◯ O(n)
- ◯ O((log(n))^2)
- ◯ O(log(n))

5

Suppose you time an algorithm for different values of N and get the results shown in the table. What is the best characterization of the asymptotic runtime complexity observed in the data? *

N	Time (s)
100	0.03
200	0.08
400	0.24
800	0.80
1600	2.87
3200	10.85
6400	42.18
12800	166.28

- ◯ O(N^3)
- ◯ O(N^2)
- ◯ O(N)
- ◯ O(log(N))
- ◯ O(1)

6

Suppose you time an algorithm for different values of N and M and get the results shown in the table. What is the best characterization of the asymptotic runtime complexity observed in the data? *

N	M	Time (s)
100	100	0.81
100	200	1.21
100	400	2.01
200	100	1.11
200	200	1.51
200	400	2.31
400	100	1.71
400	200	2.11
400	400	2.91

- ◯ O(N)
- ◯ O(M)
- ◯ O(NM)
- ◯ O(N+M)

Big-Oh for Runtime: Algorithms & Code

- What is the runtime complexity of **`stuff(n)`**?
- How many times does the loop iterate?
  - In terms of n, the parameter
- Loop body is O(1)?
  - Constant time
  - Independent of n
  - Add n same as add 1

```
7     public int stuff(int n) {
8         int sum = 0;
9         for(int k=0; k < n; k += 1) {
10            sum += n;
11        }
12        return sum;
13    }
```

Linear, O(n)

General strategy for determining Big-O runtime complexity

Most general: Determine T(N), the number of constant time operations as a function of the size of the input N. Then simplify using Big-O.

Practically, covers common cases:

1. For each line of code, label:
   a) Complexity of that line, and
   b) Number of times the line is executed
2. Add up over all lines, multiplying the two labels

Nested loop example

What about the big-O runtime complexity of this code as a function of n?

```
6       public int nested(int n) {
7           int result = 0;
8           for (int i=0; i<n; i++) {
9               for (int j=0; j<i; j++) {
10                  result += 1;
11              }
12          }
13          return result;
```

Line	Complexity	Iterations
7	O(1)	1
8	O(1)	n
9	O(1)	?
10	O(1)	?
13	O(1)	1

How many times does line 10 execute?

Nested loop example

How many times does line 10 execute?

```
6     public int nested(int n) {
7         int result = 0;
8         for (int i=0; i<n; i++) {
9             for (int j=0; j<i; j++) {
10                result += 1;
11            }
12        }
13        return result;
```

when `i` is	Line 10 executes this many times
1	1
2	2
…	…
n-2	n-2
n-1	n-1

In total? $1 + 2 + \cdots + (n-2) + (n-1) \approx \frac{n^2}{2}$ is $O(n^2)$ iterations

Nested loop example

Putting it together:

```
6       public int nested(int n) {
7           int result = 0;
8           for (int i=0; i<n; i++) {
9               for (int j=0; j<i; j++) {
10                  result += 1;
11              }
12          }
13          return result;
```

Line	Complexity	Iterations
7	O(1)	1
8	O(1)	n
9	O(1)	$O(n^2)$
10	O(1)	$O(n^2)$
13	O(1)	1

Total runtime complexity: (1) + (n) + (n^2) + (n^2) + (1)

is $O(n^2)$

Not all nested loops are quadratic

What about the big-O runtime complexity of this code as a function of n?

```
16      public int nested2(int n) {
17          int result = 0;
18          for (int i=0; i<n; i++) {
19              for (int j=0; j<100; j++)
20                  result += 1;
21              }
22          }
23          return result;
24      }
```

Line	Complexity	Iterations
17	O(1)	1
18	O(1)	n
19	O(1)	100n
20	O(1)	100n
23	O(1)	1

Total runtime complexity: (1) + (n) + (200n) + (1) is O(n)

Reminder: 200n is 200 times slower than n, but their runtimes *both scale linearly*

Not all loops are nested

What about the big-O runtime complexity of this code as a function of n?

```
28      public int parallel(int n) {
29          int result = 0;
30          for (int i=0; i<n; i++) {
31              result += 1;
32          }
33          for (int i=0; i<n; i++) {
34              result += 1;
35          }
36          return result;
37      }
```

Line	Complexity	Iterations
29	O(1)	1
30	O(1)	n
31	O(1)	n
33	O(1)	n
34	O(1)	n
36	O(1)	1

Total runtime complexity: (1) + (4n) + (1)
is O(n)

Not all loops increment by 1

Big-O Runtime complexity of calc(N) is…

- How many times does the loop iterate?
  - Concrete to abstract: calc(16), calc(32), …
- Inside loop? O(1) operations

```
143     public int calc(int n) {
144         int sum = 0;
145         for(int k=1; k < n; k *= 2) {
146             sum += k;
147         }
148         return sum;
149     }
```

Generalizing: Concrete to Abstract

N	# loop iterations
1	0
2	1
4	k=,1,2…2 iters
8	k=1,2,4…3 iters

N	# loop iterations
16	5 iterations
32	k=1,2,4,8,16..5 iters
33	6 iterations
63	6 iterations

```
143     public int calc(int n) {
144         int sum = 0;
145         for(int k=1; k < n; k *= 2) {
146             sum += k;
147         }
148         return sum;
149     }
```

O(log(N))

Accounting for iteration and non-constant time operations

What about the big-O runtime complexity of this code as a function of `n = words.size()`?

```
40      public ArrayList<String> uniqueWords (ArrayList<String> words) {
41          ArrayList<String> unique = new ArrayList<>();
42          for (String w : words) {
43              if (!unique.contains(w)) {
44                  unique.add(w);
45              }
46          }
47          return words;
48      }
```

Loop n times…

O(1), initialized empty List

O(n) in worst case

Amortized O(1), add to end

Still O(1), does not copy list to return

Total: Make n calls to O(n) `contains`: $O(n^2)$

Exponential time algorithm?

Problem from previous WOTO: What is the runtime complexity of `concatAlot` as a function of `reps`?

```
12      public static String concatAlot(int reps, String s) {
13          for (int i=0; i<reps; i++) {
14              s += s;
15          }
16          return s;
17      }
```

Loop `reps` times

Same as:
`s = s + s;`

Runtime of line 14 is O(`s.length()`). And this doubles every iteration through the loop.

Examine how the length of s grows by iterations.

Exponential time algorithm?

```
12      public static String concatAlot(int reps, String s) {
13          for (int i=0; i<reps; i++) {
14              s += s;
15          }
16          return s;
17      }
```

Loop reps times

Same as:
`s = s + s;`

Examine how the length of s grows by iterations.

Iteration	s.length()	O(1) operations (char copies)
0 (input s)	1 (suppose)	2
1	2	4
2	4	8
3	8	16
…	…	…
reps-1	2^{reps-1}	$(2)(2^{reps-1}) = 2^{reps}$

So runtime has to be at least 2^{reps}, exponential complexity!

WOTO

Go to duke.is/pu24z

Not graded for correctness, just participation.

Try to answer *without* looking back at slides and notes.

But do talk to your neighbors!

2

What is the big O runtime complexity of the keepHalving method as a function of the parameter n? *

```
102    public int keepHalving(int n) {
103        int numIterations = 0;
104        while (n > 1) {
105            n = n / 2;
106            numIterations++;
107        }
108        return numIterations;
109    }
```

- ◯ O(1)
- ◯ O(log(n))
- ◯ O(n)
- ◯ O(n^2)
- ◯ O(n^3)
- ◯ O(2^n)

3

What is the big O runtime complexity of the moreLooping method as a function of the parameter n? *

```
86    public int moreLooping(int n) {
87        int result = 0;
88        for (int i=n-1; i<n; i++) {
89            for (int k=0; k<10; k++) {
90                result += 1;
91            }
92        }
93        return result;
94    }
```

- ◯ O(1)
- ◯ O(log(n))
- ◯ O(n)
- ◯ O(n^2)
- ◯ O(n^3)
- ◯ O(2^n)

4

What is the big O runtime complexity of the reverse method as a function of n where n is the size() of the List parameter input? add(0, s) adds s to the front of the list. *

```
94    public static List<String> reverse(List<String> input) {
95        ArrayList<String> result = new ArrayList<>();
96        for (String s : input) {
97            result.add(0, s);
98        }
99        return result;
100   }
```

- ◯ O(1)
- ◯ O(log(n))
- ◯ O(n)
- ◯ O(n^2)
- ◯ O(n^3)
- ◯ O(2^n)

Runtime complexity of composed methods

- Runtime complexity of **`stuff(stuff(n))`**?

```
7     public int stuff(int n) {
8         int sum = 0;
9         for(int k=0; k < n; k += 1) {
10            sum += n;
11        }
12        return sum;
13    }
```

- Value returned by **`stuff(n)`** is n^2.
- Runtime complexity of **`stuff(n²)`**?

- **`stuff`** has linear runtime complexity, so **`stuff(n²)`** is $O(n^2)$

Composing methods general

- Given two methods:

```
public static int outer (int n) {
public static int inner(int n) {
```

- What is the runtime complexity of the following?

```
int result = outer(inner(n));
```

Running this code is equivalent to…

```
int innerValue = inner(n);
int result = outer(innerValue);
```

Composing methods general

- Given two methods:

```
public static int outer (int n) {
public static int inner(int n) {
```

- What is the runtime complexity of the following?

```
int result = outer(inner(n));
```

Three steps: Runtime complexity is 1+3.

1. Calculate runtime complexity of `inner(n)`
2. Calculate value returned by `inner(n)`
3. Calculate runtime complexity of `outer()` on value from step 2.

Composing methods example

```
int result = outer(inner(n));
```

```
56    public static int outer (int n) {
57        int result = 0;
58        for (int i=1; i<n; i*=2) {
59            result += 1;
60        }
61        return result;
62    }
63
64    public static int inner(int n) {
65        return n*n;
66    }
```

1. Runtime complexity of `inner(n)` is O(1)
2. Value returned by `inner(n)` is $O(n^2)$
3. Runtime complexity of outer(n^2) is O(log(n^2))

Recall log rules: $\log(n^2) = 2\log(n)$

Total runtime complexity: O(1) + O(log(n^2)) is O(log(n))

Most of the "work" done executing `outer`

Another composition example

```
int result = outer(inner(n));
```

```
56   public static int outer (int n) {
57       int result = 0;
58       for (int i=1; i<n; i*=2) {
59           result += 1;
60       }
61       return result;
62   }
63
64   public static int inner(int n) {
65       int result = 0;
66       for (int i=0; i<n; i++) {
67           result += n;
68       }
69       return result;
70   }
```

1. Runtime complexity of `inner(n)` is now O(n)
2. Value returned by `inner(n)` is still $O(n^2)$
3. Runtime complexity of outer(n^2) is still O(log(n^2))

Total runtime complexity: O(n) + O(log(n^2)) is O(n)

Now most of the "work" done executing `inner`