Algorithm Efficiency

CS284

Algorithm Efficiency and Big-O

- Getting a precise measure of the performance of an algorithm is difficult
- Big-O notation expresses the performance of an algorithm as a function of the number of items to be processed
- This permits algorithms to be compared for efficiency
- It does so independently of the underlying compiler
- We're going to provide an informal introduction, more in CS 385 Algorithms

Linear Growth Rate

Processing time increases in proportion to the number of inputs n

```
public static int f(int[] x, int target) {
  for (int i=0; i<x.length; i += 1) {
    if (x[i]==target) { return i; }
  }
  return -1; // target not found
}</pre>
```

Linear Growth Rate

Processing time increases in proportion to the number of inputs n

```
public static int f(int[] x, int target) {
  for (int i=0; i<x.length; i += 1) {
    if (x[i]==target) { return i; }
  }
  return -1; // target not found
}</pre>
```

- Let n be x.length
- Target not present \Rightarrow for loop will execute *n* times
- ► Target present ⇒ for loop will execute (on average) (n+1)/2 times
- Therefore, the total execution time is directly proportional to n
- This is described as a growth rate of order n or $\mathcal{O}(n)$

n * m Growth Rate

Processing time can be dependent on two different inputs n and m

```
public static boolean g(int[] x, int[] y) {
  for (int i=0; i<x.length; i += 1) {
    if (f(y, x[i]) != -1) { return false; }
  }
  return true;
}</pre>
```

n * m Growth Rate

Processing time can be dependent on two different inputs n and m

```
public static boolean g(int[] x, int[] y) {
  for (int i=0; i<x.length; i += 1) {
    if (f(y, x[i]) != -1) { return false; }
  }
  return true;
}</pre>
```

- The for loop will execute x.length times
- But it will call search, which will execute y.length times
- The total execution time is proportional to (x.length * y.length)
- The growth rate has an order of n * m or $\mathcal{O}(n * m)$

Quadratic Growth Rate

Processing time proportional to square of number of inputs n

```
public static boolean h(int[] x) {
  for (int i=0; i<x.length; i += 1) {
    for (int j=0; j<x.length; j += 1) {
        if (i != j && x[i] == x[j]) { return false; }
    }
    return true;
}</pre>
```

Quadratic Growth Rate

Processing time proportional to square of number of inputs n

```
public static boolean h(int[] x) {
  for (int i=0; i<x.length; i += 1) {
    for (int j=0; j<x.length; j += 1) {
        if (i != j && x[i] == x[j]) { return false; }
    }
    return true;
}</pre>
```

- The for loop with i as index will execute x.length times
- The for loop with j as index will execute x.length times
- The total number of times the inner loop will execute is (x.length)²
- The growth rate has an order of n^2 or $\mathcal{O}(n^2)$

Logarithmic Growth Rate

You must also examine the number of times a loop is executed

```
for (int i=1; i < x.length; i *= 2) {
   System.out.println(x[i]);
}</pre>
```

The loop body will execute k times, with i having the following values:

 $1, 2, 4, 8, 16, \dots, 2^k$

until 2^k is greater or equal to x.length

Lets deduce the value of k

$$2^{k-1} < x.length \le 2^k$$

 $\Rightarrow k-1 < \log_2(x.length) \le k \quad (\text{since } \log_2 2^k \text{ is } k)$
 $\Rightarrow k = \lceil \log_2(x.length) \rceil$

Logarithmic Growth Rate

You must also examine the number of times a loop is executed

```
for (int i=1; i < x.length; i *= 2) {
   System.out.println(x[i]);
}</pre>
```

- $k = \lceil \log_2(x.length) \rceil$
- ► Thus we say the loop is O(log₂ n)
- Logarithmic functions grow slowly as the number of data items n increases

Different Growth Rates



Growth Rate

Defining Big-O

Big-O Notation

- The O() in the previous examples can be thought of as an abbreviation of "order of magnitude"
- $f(n) \in \mathcal{O}(g(n))$
- ► We can thus say that g(n) is an upper bound on the growth rate
- We are next going to define $\mathcal{O}()$ more precisely

Formal Definition of Big-O

- Consider the two snippets of code below
- In order to compare their growth rates, why not just count the number of time units for each?

```
for (int i = 0; i < n; i += 1) {
  for (int j = 0; j < 7; j += 1) {
    System.out.println("Hello");
  }
  for (int j = 0; j < 50; j += 1) {
    System.out.println("Hello");
  }
}
</pre>
for (int i = 0; i < n; i += 1) {
  for (int j = 0; j < 100; j += 1)
    System.out.println("Hello");
  }
}
```

 $\mathcal{T}_1(n) = 7n + 50 \qquad \qquad \mathcal{T}_2(n) = 100n$

- ▶ For large values of *n* independent terms (such as 50) and constant coefficients (such as 7 and 100) are negligible
- Both are considered to have linear growth

Formal Definition of Big-O

 $\mathcal{O}(g(n)) = \{f(n) \mid \text{ there exist two positive constants, } n_0 \text{ and } c \text{ such that, } 0 \leq f(n) \leq c * g(n) \text{ for all } n \geq n_0 \}$

• $\mathcal{O}(g(n))$ is a set of functions

► It is the set of functions f(n) s.t., as n gets sufficiently large (larger than n₀), there is some constant c for which the processing time will always be less than or equal to c * g(n)

$$n^2+5n+25\in\mathcal{O}(n^2)$$

Find constants n_0 and c so that, for all $n > n_0$, $cn^2 \ge n^2 + 5n + 25$

1

$$cn^{2} \geq n^{2} + 5n + 25$$

$$c \geq \frac{n^{2}}{n^{2}} + \frac{5n}{n^{2}} + \frac{25}{n^{2}}$$

$$c \geq 1 + \frac{5}{n} + \frac{25}{n^{2}}$$

- When $n = n_0 = 5$, the RHS is $(1 + \frac{5}{5} + \frac{25}{25})$, c is 3
- Moreover, $\lim_{n\to\infty} 1 + \frac{5}{n} + \frac{25}{n^2} = 1$
- So, $3n^2 \ge n^2 + 5n + 25$, for all $n \ge 5$
- Other values of n_0 and c also work



Consider the following loop

$$T(n) = 3(n-1) + 3(n-2) + ... + 3$$

Question:

 $\mathcal{T}(n) \in \mathcal{O}(n^2)$?

$$T(n) = 3(n-1) + 3(n-2) + ... + 3$$

Factoring out the 3,
$$3(n-1+n-2+...+1)$$

$$1+2+...+n-1 = \frac{(n*(n-1))}{2}$$
Therefore $\mathcal{T}(n) = 1.5n^2 - 1.5n$

$$cn^2 \geq 1.5n^2 - 1.5n$$

$$c \geq 1.5 - \frac{1.5}{n_0}, n_0 \geq 1$$

• Therefore $\mathcal{T}(n) \in \mathcal{O}(n^2)$ when n_0 is 1 and c is 1.5



Exercises

- Show that $\mathcal{T}(n) = n^3 5n^2 + 20n 20 \in \mathcal{O}(n^3)$.
- Show that $T(n) = 7n^4 + 5n^2 50n \in O(n^4)$.

Symbols Used in Quantifying Performance

Symbol	Meaning
T(n)	The time that a method or program takes as a function of the number of inputs, n . We may not be able to measure or determine this exactly.
f(<i>n</i>)	Any function of <i>n</i> . Generally, $f(n)$ will represent a simpler function than $T(n)$, for example, n^2 rather than $1.5n^2 - 1.5n$.
O (f(<i>n</i>))	Order of magnitude. $O(f(n))$ is the set of functions that grow no faster than $f(n)$. We say that $T(n) = O(f(n))$ to indicate that the growth of $T(n)$ is bounded by the growth of $f(n)$.

Common Growth Rates

Big-O	Name
$\mathcal{O}(1)$	Constant
$\mathcal{O}(\log n)$	Logarithmic
$\mathcal{O}(n)$	Linear
$\mathcal{O}(n \log n)$	Log-linear
$\mathcal{O}(n^2)$	Quadratic
$\mathcal{O}(n^3)$	Cubic
$\mathcal{O}(2^n)$	Exponential
$\mathcal{O}(n!)$	Factortial

Effects of Different Growth Rates

O(f(<i>n</i>))	f(50)	f(100)	f(100)∕f(50)
O (1)	1	1	1
$O(\log n)$	5.64	6.64	1.18
O (<i>n</i>)	50	100	2
$O(n \log n)$	282	664	2.35
$O(n^2)$	2500	10,000	4
O (<i>n</i> ³)	12,500	100,000	8
O(2 ⁿ)	1.126×10^{15}	1.27×10^{30}	1.126×10^{15}
O(<i>n</i> !)	3.0×10^{64}	9.3×10^{157}	3.1×10^{93}

Algorithms with Exponential and Factorial Growth Rates

- Algorithms with exponential and factorial growth rates have an effective practical limit on the size of the problem they can be used to solve
- With an $\mathcal{O}(2^n)$ algorithm, if 100 inputs takes an hour then,
 - 101 inputs will take 2 hours
 - 105 inputs will take 32 hours
 - 114 inputs will take 16,384 hours (almost 2 years!)

Algorithms with Exponential and Factorial Growth Rates (cont.)

- Encryption algorithms take advantage of this characteristic
- Some cryptographic algorithms can be broken in O(2ⁿ) time, where n is the number of bits in the key
- A key length of 40 is considered breakable by a modern computer, but a key length of 100 bits will take a billion-billion (1018) times longer than a key length of 40