Data Structures

Algorithm Efficiency

CS284

Objectives

- Algorithm efficiency
- Big-O notation for measuring algorithm efficiency
- Comparing efficiency

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*

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

Linear Growth Rate

Processing time increases in proportion to the number of inputs n

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

- Let n be `x.length`
- Target not present $\Rightarrow$ for loop will execute n times
- Target present $\Rightarrow$ 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 areDifferent(int[] x, int[] y) {
  for(int i=0; i<x.length; i++) {
    if (search(y, x[i]) != -1)
      return false;
  }
  return true;
}
```

$n * m$ Growth Rate

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

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

- 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 areUnique(int[] x) {
  for(int i=0; i<x.length; i++) {
    for(int j=0; j<x.length; j++) {
      if (i != j && x[i] == x[j])
        return false;
    }
  }
  return true;
}
```

Quadratic Growth Rate

Processing time proportional to square of number of inputs n

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

- 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 $\mathcal{O}(n^2)$

Big-O Notation

- The $\mathcal{O}()$ in the previous examples can be thought of as an abbreviation of "order of magnitude"
- A simple way to determine the big-O notation of an algorithm is to look at the loops and to
  - see whether the loops are nested
  - consider the number of times a loop is executed
- Assuming a loop body consists only of simple statements and executes at most n times,
  - a single loop is $\mathcal{O}(n)$
  - a pair of nested loops is $\mathcal{O}(n^2)$
  - a nested pair of loops inside another is $\mathcal{O}(n^3)$
  - and so on . . .

Big-O Notation

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

```
for(int i=1; i < x.length; i *= 2) {
   // Do something with x[i]
}
```

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

$$1, 2, 4, 8, 16, ..., 2^k$$

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

- Lets deduce the value of k

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

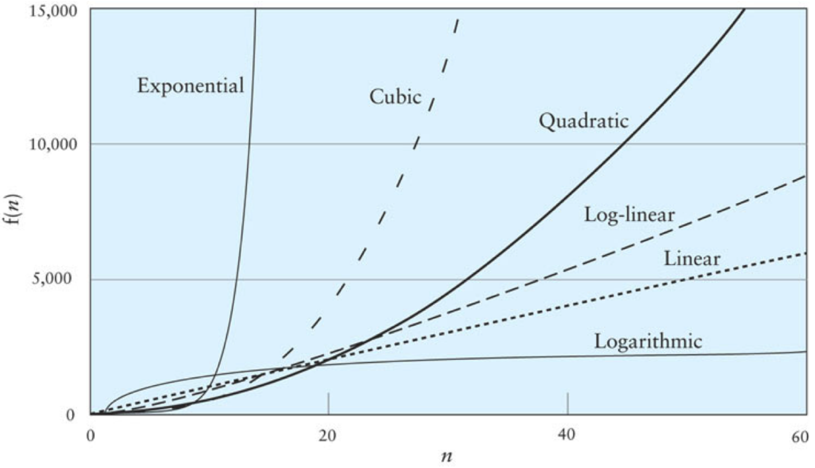
Big-O Notation

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

```
for(int i=1; i < x.length; i *= 2) {
   // Do something with x[i]
}
```

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

Different Growth Rates



Formal Definition of Big-O

- Nested loop executes Simple Statement n^2 times
- Loop executes 5 Simple Statements n times
- 25 Simple Statements are executed
- Conclusion: the relationship between processing time and n (number of data items processed) is: $\mathcal{T}(n) = n^2 + 5n + 25$

```
for (int i = 0; i < n; i++) {
 for (int j = 0; j < n; j++) {
          Simple Statement
  }
}
for (int i = 0; i < n; i++) {
   Simple Statement 1
   Simple Statement 2
   Simple Statement 3
   Simple Statement 4
   Simple Statement 5
}
Simple Statement 6
Simple Statement 7
...
Simple Statement 30
```

Why Not Compare Algorithms Using $\mathcal{T}$?

- Comparing algorithms based on running time $\mathcal{T}(n)$ is not convenient
- The formulas describing $\mathcal{T}$ can be complicated
- It is better to bound $\mathcal{T}$ by another function
- That way algorithms can be compared by comparing their bounds
- The Big-o notation $\mathcal{O}(f(n))$ for $\mathcal{T}$ means "roughly" that $f(n)$ is a bound for $\mathcal{T}$, for large enough values of n
- Lets make this more precise

Formal Definition of Big-O

$\mathcal{T}(n) =$ time it takes an algorithm to process n data items

- In terms of $\mathcal{T}(n)$,

$$\mathcal{T}(n) = \mathcal{O}(f(n))$$

  - means that there exist two constants, n_0 and c, greater than zero such that for all $n > n_0$,

$$\mathcal{T}(n) \leq cf(n)$$

- In other words, as n gets sufficiently large (larger than n_0), there is some constant c for which the processing time will always be less than or equal to $cf(n)$
- $cf(n)$ is an upper bound on performance

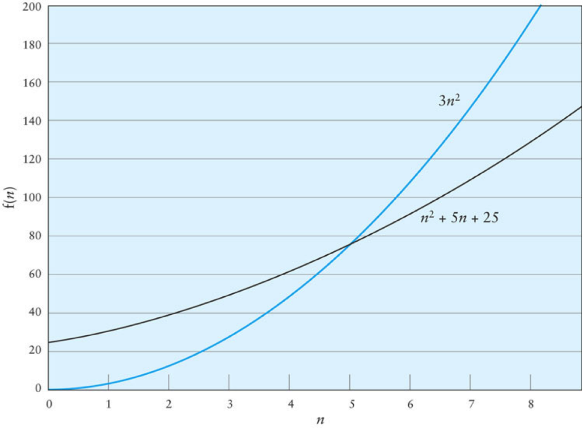
Formal Definition of Big-O

- The growth rate of $f(n)$ will be determined by the fastest growing term, which is the one with the largest exponent
- In the example, $\mathcal{T}(n) = n^2 + 5n + 25$ has growth order $\mathcal{O}(n^2)$
- In general, it is safe to ignore all constants and to drop the lower-order terms when determining the order of magnitude

Big-O Example 1

- Given $\mathcal{T}(n) = n^2 + 5n + 25$, show that this is $\mathcal{O}(n^2)$
- Find constants n_0 and c so that, for all $n > n_0$, $cn^2 > n^2 + 5n + 25$
  - Find the point where $cn^2 = n^2 + 5n + 25$
  - Let $n = n_0$, and solve for c, $c = 1 + \frac{5}{n_0} + \frac{25}{n_0^2}$
- When n_0 is 5, the RHS is $(1 + \frac{5}{5} + \frac{25}{25})$, c is 3
- So, $3n^2 > n^2 + 5n + 25$, for all $n > 5$
- Other values of n_0 and c also work

Big-O Example 1



Big-O Example 2

- Consider the following loop

```
for (int i = 0; i < n; i++)
{
  for (int j = i + 1; j < n; j++)
  {
    3 simple statements
  }
}
```

$$\mathcal{T}(n) = 3(n-1) + 3(n-2) + ... + 3$$

- Lets compute a bound on the growth rate

Big-O Example 2 (cont.)

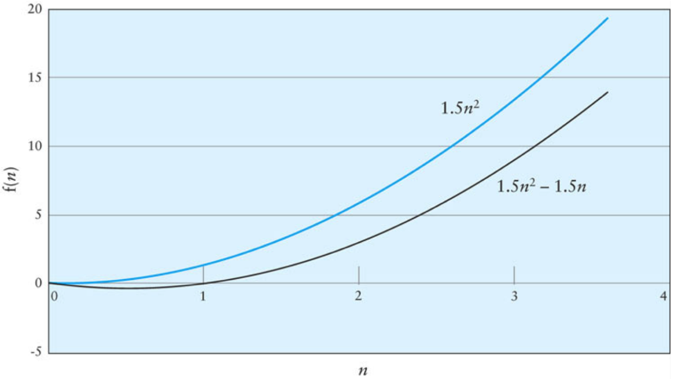
$$\mathcal{T}(n) = 3(n-1) + 3(n-2) + ... + 3$$

- Factoring out the 3,

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

- $1 + 2 + ... + n - 1 = \frac{(n*(n-1))}{2}$
- Therefore $\mathcal{T}(n) = 1.5n^2 - 1.5n$
- When $n = 1$, the polynomial has value 0
- For values of $n > 1$, $1.5n^2 > 1.5n^2 - 1.5n$
- Therefore $\mathcal{T}(n)$ is $\mathcal{O}(n^2)$ when n_0 is 1 and c is 1.5

Big-O Example 2



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.
$f(n)$	Any function of n. Generally, $f(n)$ will represent a simpler function than $T(n)$, for example, n^2 rather than $1.5n^2 - 1.5n$.
$\mathbf{O}(f(n))$	Order of magnitude. $\mathbf{O}(f(n))$ is the set of functions that grow no faster than $f(n)$. We say that $T(n) = \mathbf{O}(f(n))$ 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

$O(f(n))$	$f(50)$	$f(100)$	$f(100)/f(50)$
$O(1)$	1	1	1
$O(\log n)$	5.64	6.64	1.18
$O(n)$	50	100	2
$O(n \log n)$	282	664	2.35
$O(n^2)$	2500	10,000	4
$O(n^3)$	12,500	100,000	8
$O(2^n)$	1.126×10^{15}	1.27×10^{30}	1.126×10^{15}
$O(n!)$	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 $\mathcal{O}(2^n)$ 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