Algorithm Efficiency and More List implementations

Algorithm Efficiency and Big-O

Section 2.4

+ 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
- For more than a certain number of data items, some problems cannot be solved by any computer

+ Linear Growth Rate

- If processing time increases in proportion to the number of inputs *n*, the algorithm grows at a linear rate

```
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

- If the target is not present, the `for` loop will execute `x.length` times
- If the target is present the `for` loop will execute (on average) `(x.length + 1)/2` times
- Therefore, the total execution time is directly proportional to `x.length`
- This is described as a growth rate of order *n* OR
- O(n)

- If processing time in… s *n*, the algorithm grow…

```
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
}
```

+ n x m Growth Rate

- Processing time can be dependent on two different inputs

```
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 x m Growth Rate (cont.)

- Processing time can b

- 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 x m or
- O(n x 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;
}
```

+ Quadratic Growth Rate

- If processing time is proportional to the square of the number of inputs n, the algorithm grows at a quadratic rate (n^2)

```
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 (cont.)

- 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 $(\texttt{x.length})^2$
- The growth rate has an order of n^2 or
- $O(n^2)$

- If processing time is p... inputs n, the algorithm...

```
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;
}
```

+ Big-O Notation

- The 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
- Assuming a loop body consists only of simple statements,
  - a single loop is $O(n)$
  - a pair of nested loops is $O(n^2)$
  - a nested pair of loops inside another is $O(n^3)$
  - and so on . . .

+ **Big-O Notation** (cont.)

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

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

- The loop body will execute k-1 times, with `i` having the following values:
  1, 2, 4, 8, 16, . . ., 2^k
  until 2^k is greater than `x.length`

- Since 2^{k-1} = `x.length` < 2^k and $\log_2 2^k$ is k, we know that k-1 = $\log_2$(`x.length`) < k

- Thus we say the loop is O(log n) (in analyzing algorithms, we use logarithms to the base 2)

- Logarithmic functions grow slowly as the number of data items n increases

+ **Formal Definition of Big-O**

- Consider the following program structure:

```
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
```

+ Formal Definition of Big-O (cont.)

- Consider the following program structure:

```
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
```

This nested loop executes a *Simple Statement* n^2 times

+ Formal Definition of Big-O (cont.)

- Consider the following program structure:

```
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
```


This loop executes 5 *Simple Statements* n times ($5n$)

+ Formal Definition of Big-O (cont.)

- Consider the following program structure:

```
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
```

Finally, 25 *Simple Statements* are executed

+ Formal Definition of Big-O (cont.)

- Consider the following program structure:

```
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
```


We can conclude that the relationship between processing time and n (the number of date items processed) is:

$$T(n) = n^2 + 5n + 25$$

+ Formal Definition of Big-O (cont.)

- In terms of $T(n)$,

$$T(n) = O(f(n))$$

- There exist
  - two constants, n_0 and c, greater than zero, and
  - a function, $f(n)$,
- such that for all $n > n_0$, $cf(n) = T(n)$
- 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 (cont.)

- 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, an algorithm of

$$O(n^2 + 5n + 25)$$

  is more simply expressed as

$$O(n^2)$$

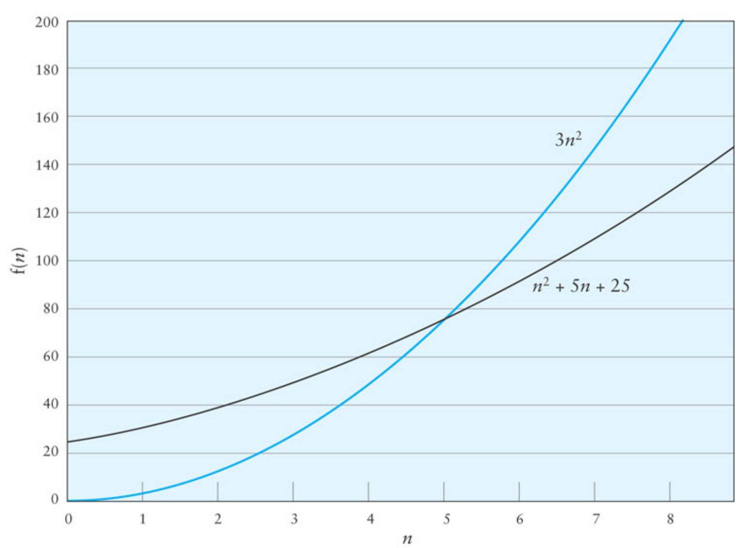
- In general, it is safe to ignore all constants and to drop the lower-order terms when determining the order of magnitude

+ Big-O Example 1

- Given T(n) = $n^2 + 5n + 25$, show that this is $O(n^2)$
- 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

    $c = 1 + 5/n_0, + 25/n_0^2$
- When n_0 is 5(1 + 5/5 + 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 (cont.)



+ Big-O Example 2

- Programming problem 1 shows
```
y1 = 100 * n + 10;
y2 = 5 * n * n + 2;
```
- It asks to write a program that compares y1 and y2 for values of n up to 100 in increments of 10
- Before writing the code, at what value of n will y2 consistently be greater than y1?
- How does this relate to the problem:
  - T(n) = 5 * n * n + 2 + 100 * n + 10
  - for what values of n0 and c n*n consistently larger than T(n)

+ Big-O Example 3

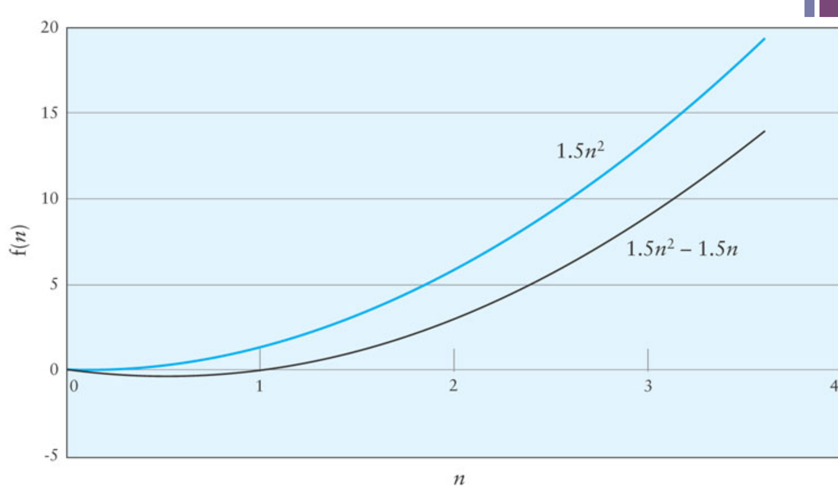
- Consider the following loop
```
for (int i = 0; i < n; i++) {
  for (int j = i + 1; j < n; j++) {
     3 simple statements
  }
}
```
- $T(n) = 3(n – 1) + 3 (n – 2) + … + 3$
- Factoring out the 3,

  $3(n – 1 + n – 2 + n – 3 + … + 1)$
- $1 + 2 + … + n – 1 = (n \times (n-1))/2$

+ Big-O Example 3 (cont.)

- Therefore $T(n) = 1.5n^2 - 1.5n$
- When $n = 0$, the polynomial has the value 0
- For values of $n > 1$, $1.5n^2 > 1.5n^2 - 1.5n$
- Therefore $T(n)$ is $O(n^2)$ when n_0 is 1 and c is 1.5

+ Big-O Example 2 (cont.)



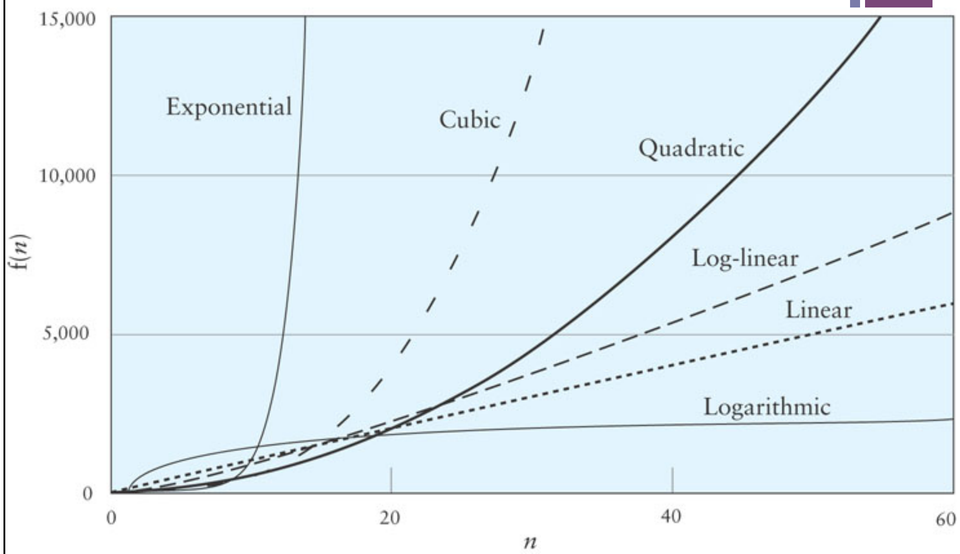
+ Symbols Used in Quantifying Performance

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 $1.5n^2 - 1.5n$.
$O(f(n))$	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
$O(1)$	Constant
$O(\log n)$	Logarithmic
$O(n)$	Linear
$O(n \log n)$	Log-linear
$O(n^2)$	Quadratic
$O(n^3)$	Cubic
$O(2^n)$	Exponential
$O(n!)$	Factorial

+ Different Growth Rates



+ 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 $O(2^n)$ 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 $O(2^n)$ 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 (10^{18}) times longer than a key length of 40

ArrayList implementation

- Version One: Fixed array
- Version Two: Making it Generic
- Version Three: Dynamic Array

Simplified List Interface (ADT)

```
public interface SimplifiedList {
        public boolean add(Object item);
        public boolean add(int index, Object item);

        public Object remove(int index);
        public Object set(int index, Object item);
        public Object get(int index);

        public boolean contains (Object item);

        public boolean isEmpty();
        public void clear();
        public int size();

        public boolean isFull();
 } // interface SimplifiedList
```

+ Implementing SimplifiedList

- Data Structure#1: fixed size array (capacity)
  - Use an array of Object to store stuff
  - A variable to store # entries in it
  - Its capacity (max size)

+ Version 2: Making it Generic

- Change the interface and the class

Simplified List Interface (ADT) <Generic>

```
public interface SimplifiedList<E> {
        public boolean add(E item);
        public boolean add(int index, E item);

        public E remove(int index);
        public E set(int index, E item);
        public E get(int index);

        public boolean contains (E item);

        public boolean isEmpty();
        public void clear();
        public int size();

        public boolean isFull();
 } // interface SimplifiedList
```

Version 3: Using flexible arrays

- add a method called void reallocate() that
  - is called when adding to an array that is at capacity
  - doubles the size of the array copying the current values to the new values.

```
private void reallocate() {
   capacity = 2 * capacity;
   theData = Arrays.copyOf(theData, capacity);
}
```

+ Performance of ArrayLists

	Fixed Size Array	Dynamic Array
add(o)	O(1)	O(n)
add(i, o)	O(n)	O(n)
remove(i)	O(n)	O(n)
set(i, o)	O(1)	O(1)
get(i)	O(1)	O(1)
contains(o)	O(n)	O(n)
clear(), size(), isEmpty()	O(1)	O(1)