CSE214 Data Structures

Order of Complexity

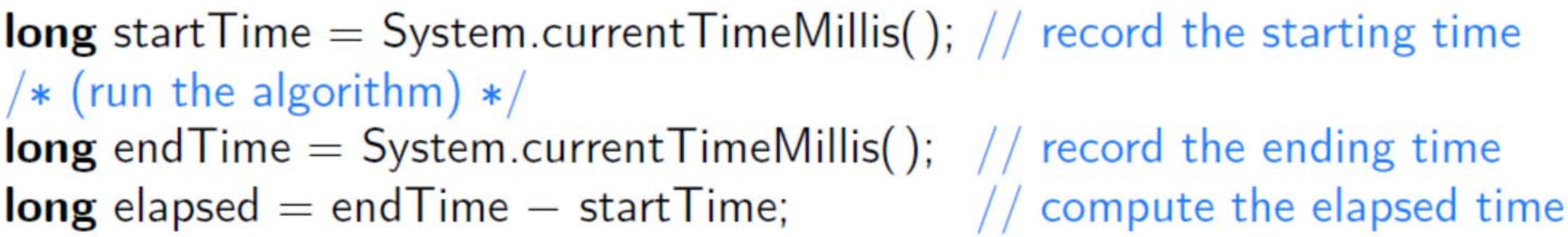
YoungMin Kwon

Analysis Tools

- Data structure
  - A systematic way of organizing and accessing data
- Algorithm
  - A step-by-step procedure for performing some tasks in a finite amount of time
- Analysis tool
  - A measure that can tell how “good” a data structure or an algorithm is
  - Running time
  - Space usage

Empirical Analysis

- Record the running time
    - currentTimeMillis(): the number of milliseconds since 1/1/1970 UTC.
    - nanoTime(): for more accurate measurements



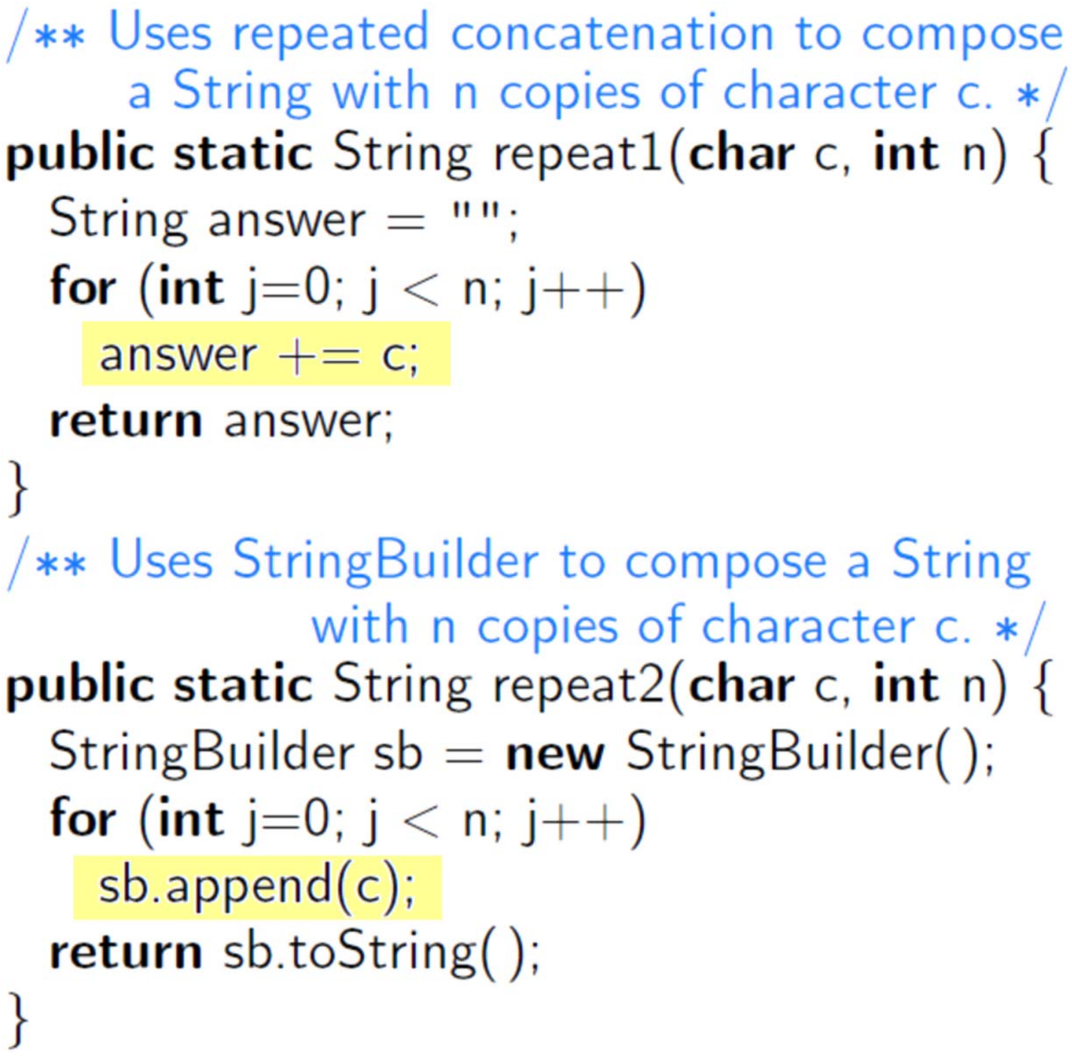
```
long startTime = System.currentTimeMillis(); // record the starting time
/* (run the algorithm) */
long endTime = System.currentTimeMillis();   // record the ending time
long elapsed = endTime − startTime;          // compute the elapsed time
```

Empirical Analysis

- Measure the running-time many times
  - With respect to varying input size and structure
  - Plot the running-time
  - Find the best fitting function through statistical analyses

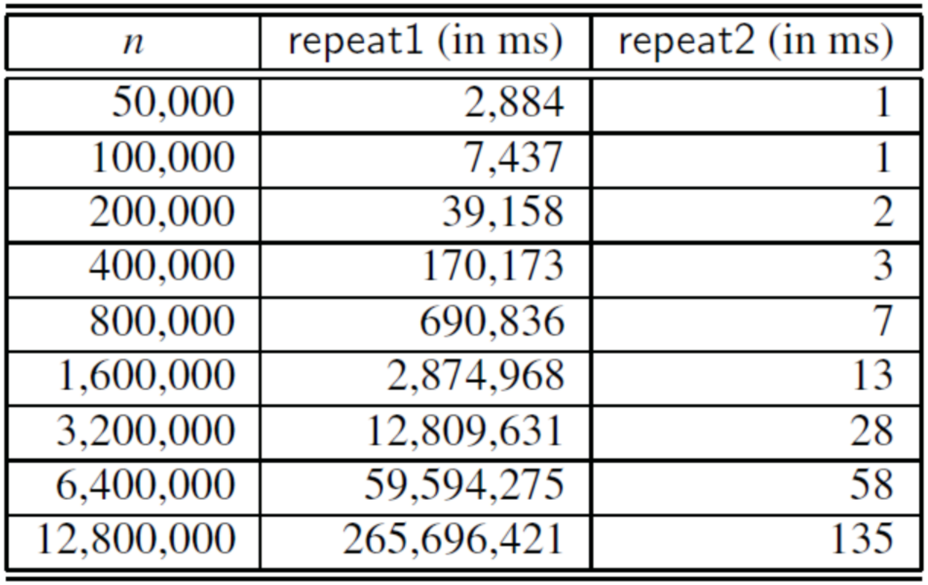
- Challenges of experimental analysis
  - The measured time may vary depending on the environments (computer's CPU power, amount of memory, what other processes are running concurrently)
  - Limited set of test inputs
  - An algorithm must be fully implemented

Empirical Analysis (Example)

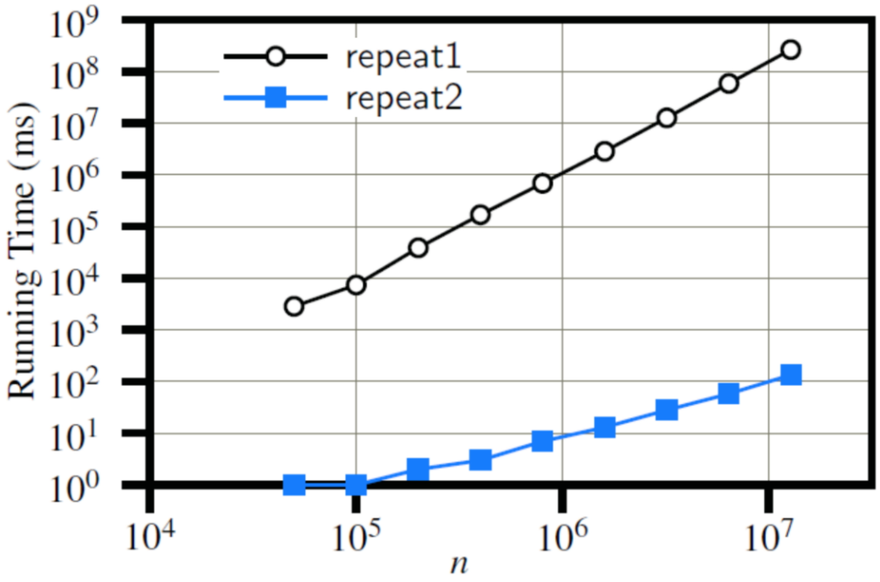


```
/** Uses repeated concatenation to compose
      a String with n copies of character c. */
public static String repeat1(char c, int n) {
  String answer = "";
  for (int j=0; j < n; j++)
    answer += c;
  return answer;
}
/** Uses StringBuilder to compose a String
                  with n copies of character c. */
public static String repeat2(char c, int n) {
  StringBuilder sb = new StringBuilder( );
  for (int j=0; j < n; j++)
    sb.append(c);
  return sb.toString( );
}
```

Empirical Analysis (Example)



n	repeat1 (in ms)	repeat2 (in ms)
50,000	2,884	1
100,000	7,437	1
200,000	39,158	2
400,000	170,173	3
800,000	690,836	7
1,600,000	2,874,968	13
3,200,000	12,809,631	28
6,400,000	59,594,275	58
12,800,000	265,696,421	135



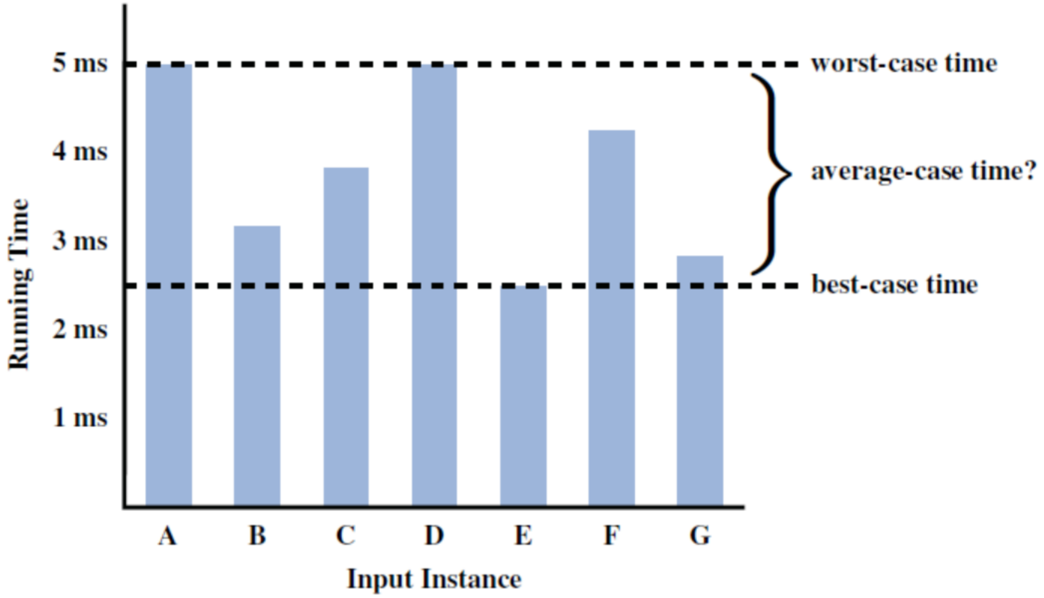
- To compose 12,800,000 strings
  - repeat1 takes more than 3 days
  - repeat2 takes less than a second

Beyond Experimental Analysis

- Desirable properties
  - Independent of h/w or s/w environments
  - No need for implementations
  - Account for all possible inputs
- Counting primitive operations
  - E.g. assignment, following an object reference, arithmetic operation, indexing an array element, calling a method, returning from a method
  - Associate a function $f(n)$: the count of primitive operations in terms of the input size n

Beyond Experimental Analysis

- Focusing on the worst-case input
  - Average-case analysis: need a probability distribution on the set of inputs (difficult to obtain)
  - Worst-case analysis: if an algorithm works well on the worst-case, it works well on every input



Common Mathematical Functions

- Constant function
  - $f(n) = c$
  - E.g. a sequential block of code
- Linear function
  - $f(n) = n$
  - E.g. finding the max from an array

Common Mathematical Functions

- Logarithm function
  - $f(n) = \log_b n$
  - $x = \log_b n$ if and only if $b^x = n$
  - For CS, base b=2 is typical and we will omit it
  - E.g. binary search on a sorted array

- Logarithm rules
  - $\log_b(a \cdot c) = \log_b a + \log_b c$
  - $\log_b(a / c) = \log_b a - \log_b c$
  - $\log_b(a^c) = c \cdot \log_b a$
  - $\log_b a = \log_d a / \log_d b$
  - $b\text{^}\log_d a = a\text{^}\log_d b$

Common Mathematical Functions

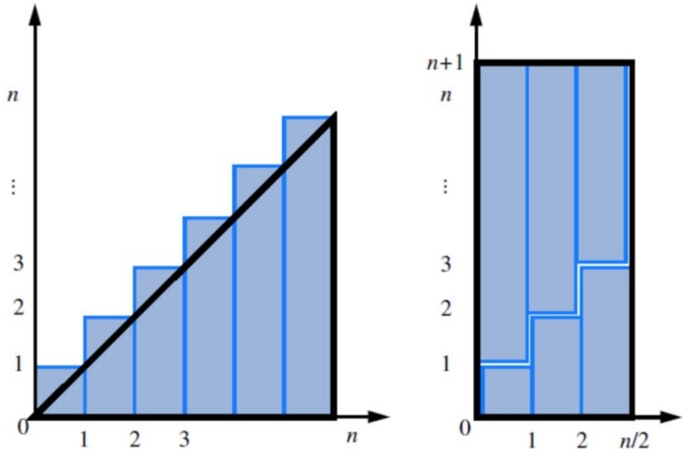
- *N-log-N* function
  - $f(n) = n \cdot \log n$
  - E.g. fast sorting algorithms such as quick sort

Common Mathematical Functions

- Quadratic function
  - $f(n) = n^2$
  - E.g. nested loop

```
for( i = 0; i < n; i++ )
    for( j = 0; j < i; j++ )
        /*do something*/
```

$$1+2+3+\cdots+(n-2)+(n-1)+n=\frac{n(n+1)}{2}$$



Common Mathematical Functions

- Cubic function
    - $f(n) = n^3$
- Polynomials
    - $f(n) = a_0 + a_1 n + a_2 n^2 + \ldots + a_d n^d$
    - $a_0, a_1, a_2, \ldots, a_d$ are coefficients and $a_d \neq 0$
    - d is the degree of the polynomial

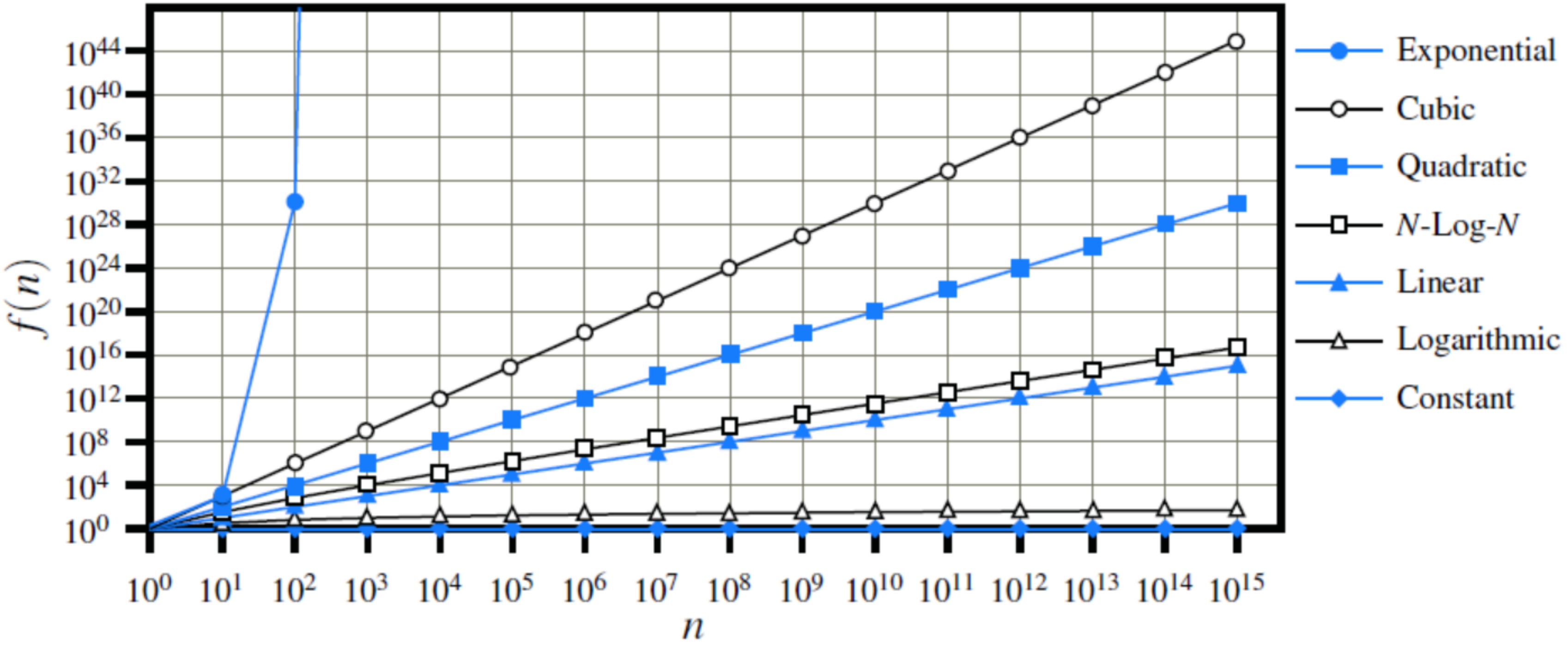
Common Mathematical Functions

- Exponential function
  - $f(n) = b^n$
  - b is a positive constant called base, n is the exponent
- Exponent rules
  - $(b^a)^c = b^{ac}$
  - $b^a b^c = b^{(a+c)}$
  - $b^a / b^c = b^{(a-c)}$
- Geometric summation

$$\sum_{i=0}^{n} a^i = 1 + a + a^2 + \cdots + a^n = \frac{a^{n+1} - 1}{a - 1}$$

Comparing Growth Rates

constant	**logarithm**	**linear**	***n*-log-*n***	**quadratic**	**cubic**	**exponential**
1	$\log n$	n	$n \log n$	n^2	n^3	a^n



Big-Oh Notation

- Analyze algorithms using a mathematical function that disregards constant factors
  - Each basic step in a pseudocode or a high level language implementation → a small number of primitive operations
  - Make analyses independent to language-specific or hardware-specific details

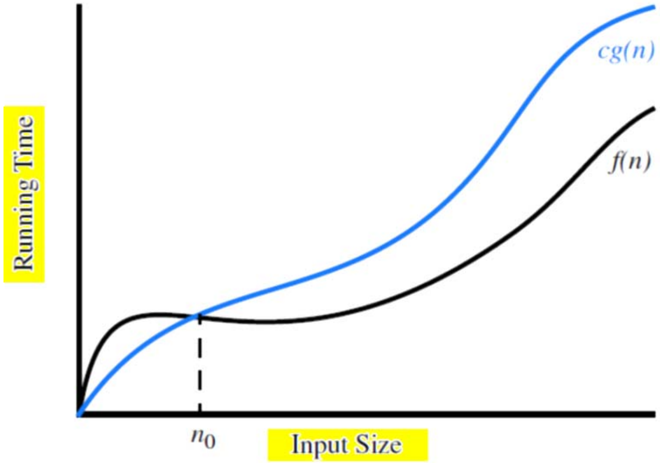
  -
    ```
    for( i = 0; i < n; i++ )
        for( j = 0; j < i; j++ )
            a = i + j; /*constant part*/
    ```

Big-Oh Notation (Definition)

- Let $f(n)$ and $g(n)$ be functions from positive integers to real numbers.
- We say that $f(n)$ is $O(g(n))$ if there is a positive real constant c and an integer constant $n_0 \geq 1$ such that

$$f(n) \leq c \cdot g(n), \quad for\ n \geq n_0$$

  - $c \cdot g(n)$ is an asymptotic upper bound



The function $f(n)$ is $O(g(n))$, since $f(n) \leq c \cdot g(n)$ when $n \geq n_0$.

Big-Oh Notation

- $f(n) = O(g(n))$
  - $f(n)$ is less than or equal to $g(n)$ up to a constant factor in the asymptotic sense as n grows towards infinity

- Example
  - $8 \cdot n + 5$ is $O(n)$
    $8 \cdot n + 5 \leq c \cdot n$ for $c = 9$ and $n_0 = 5$
  - There can be other choices of c and n_0 such as $c = 13$ and $n_0 = 1$

Big-Oh Notation

- Big-Oh notation allows us to ignore constant factors and lower-order terms

- If $f(n)$ is a polynomial of degree d,

$$f(n) = a_0 + a_1 n + \ldots + a_d n^d$$

then $f(n)$ is $O(n^d)$

  - $g(n) = n^d$,
  - $c = |a_0| + |a_1| + \ldots + |a_d|$,
  - $n_0 = 1$

Big-Oh Notation

- Use big-Oh notation to characterize a function as closely as possible
    - $f(n) = 4n^3 + 3n^2$ can be $O(n^5)$, $O(n^4)$, or $O(n^3)$, but it is more accurate to say that $f(n)$ is $O(n^3)$
- It is considered poor taste to include constant factors and lower order terms
    - It is not fashionable to say that $f(n) = 4n^3 + 3n^2$ is $O(5n^3)$ or $O(4n^3 + 4n^2)$
- The 7 common mathematical functions are the most commonly used functions for the big-Oh notation

Big-Omega Notation

- Big-Omega
  - Asymptotic way of saying that a function grows at a rate greater than or equal to that of another
  - $f(n)$ is $\Omega(\, g(n)\,)$ if $g(n)$ is $O(\, f(n)\,)$: there is a positive real constant c and an integer constant $n_0 \geq 1$ s.t.

    $$f(n) \geq c \cdot g(n), \quad \text{for } n \geq n_0$$

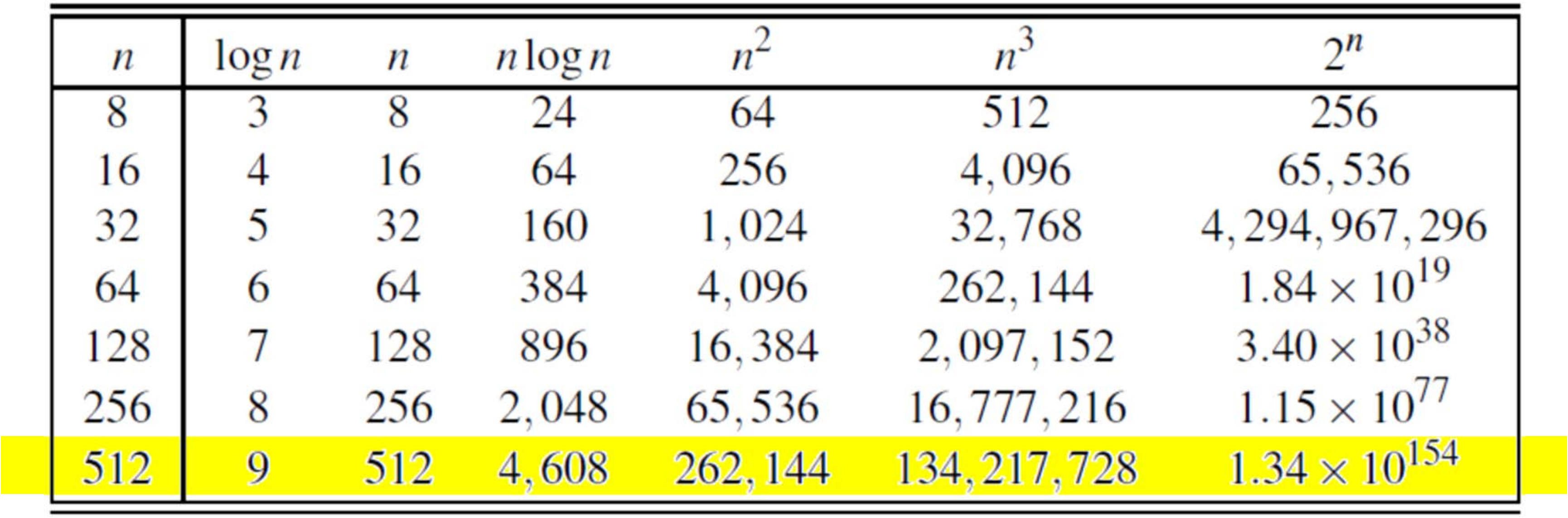
    - $c \cdot g(n)$ is an asymptotic lower bound
  - Example
    - $3 \cdot n \cdot \log n - 2 \cdot n$ is $\Omega(n \cdot \log n)$

Big-Theta Notation

- Big-Theta
  - $f(n)$ is $\Theta(\, g(n)\,)$ if $f(n)$ is both $O(\, g(n)\,)$ and $\Omega\,(\, g(n)\,)$
  - $f(n)$ is $\Theta(\, g(n)\,)$ if there are positive real constants c' and c'' and an integer constant $n_0 \geq 1$ s.t.
    $c' \cdot g(n) \leq f(n) \leq c'' \cdot g(n), \quad for\ n \geq n_0$
  - Example
    $3 \cdot n \cdot \log n - 2 \cdot \mathrm{n}$ is $\Theta(n \cdot \log n)$

Comparative Analysis

- Big-Oh notations are widely used for running-time and space bounds in terms of input size
  - Asymptotically slower algorithms are beaten in the long run by asymptotically faster algorithms



n	$\log n$	n	$n \log n$	n^2	n^3	2^n
8	3	8	24	64	512	256
16	4	16	64	256	4,096	65,536
32	5	32	160	1,024	32,768	4,294,967,296
64	6	64	384	4,096	262,144	1.84×10^{19}
128	7	128	896	16,384	2,097,152	3.40×10^{38}
256	8	256	2,048	65,536	16,777,216	1.15×10^{77}
512	9	512	4,608	262,144	134,217,728	1.34×10^{154}

Comparative Analysis

- Inefficient algorithms
  - Draw a line between polynomial time algorithms and exponential time algorithms
    - Exponential time algorithms are not considered tractable.
  - Note: $O(n^{100})$ is a polynomial time algorithm, but should not be considered efficient

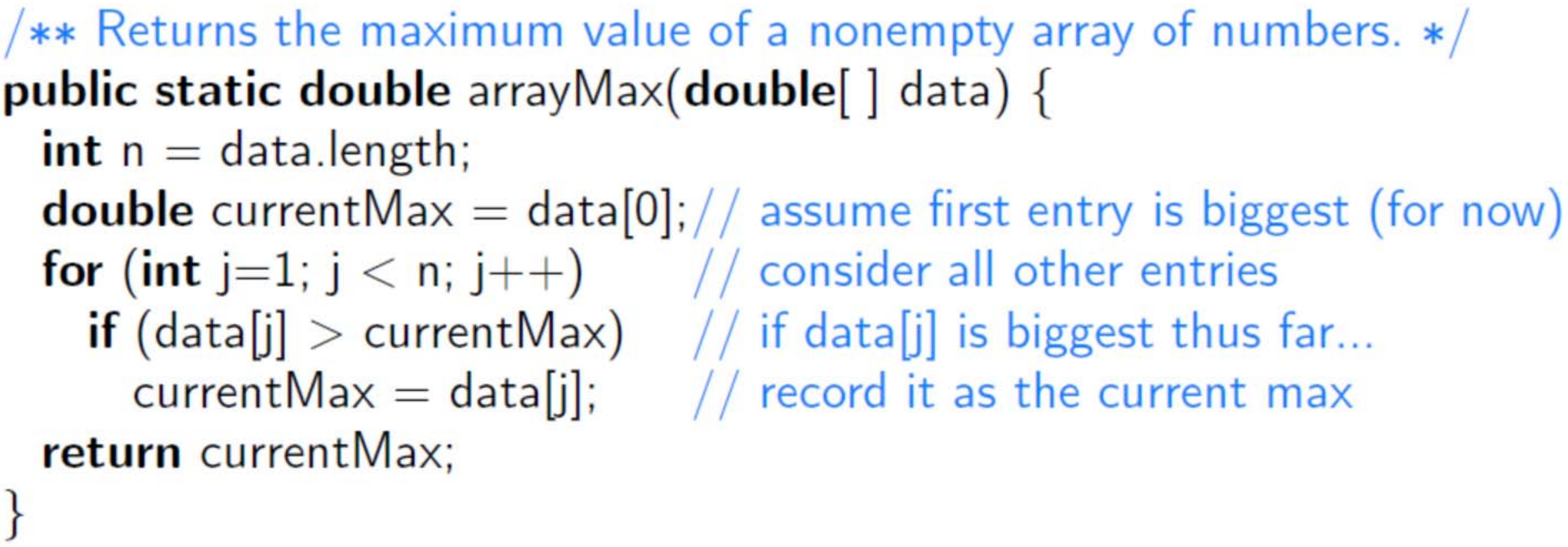
Examples of Algorithm Analysis

- O(1): constant-time operations
  - Primitive operations

```
a = a + 1;
a = arr[i];
a = arr.length;
```

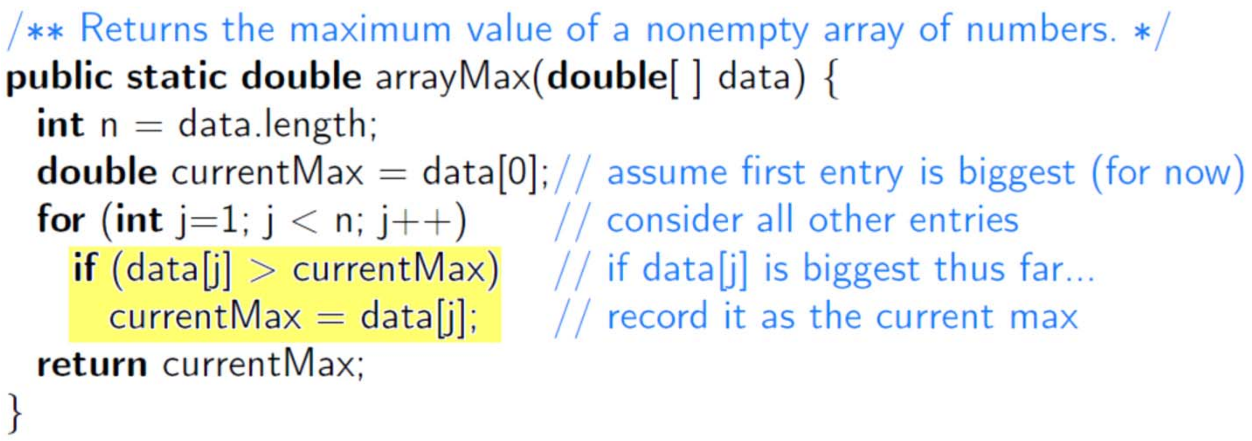
Examples of Algorithm Analysis

- Find the max of an array



```
/** Returns the maximum value of a nonempty array of numbers. */
public static double arrayMax(double[ ] data) {
  int n = data.length;
  double currentMax = data[0];// assume first entry is biggest (for now)
  for (int j=1; j < n; j++)      // consider all other entries
    if (data[j] > currentMax)    // if data[j] is biggest thus far...
      currentMax = data[j];      // record it as the current max
  return currentMax;
}
```

Examples of Algorithm Analysis



```
/** Returns the maximum value of a nonempty array of numbers. */
public static double arrayMax(double[ ] data) {
  int n = data.length;
  double currentMax = data[0];// assume first entry is biggest (for now)
  for (int j=1; j < n; j++)      // consider all other entries
    if (data[j] > currentMax)    // if data[j] is biggest thus far...
      currentMax = data[j];      // record it as the current max
  return currentMax;
}
```

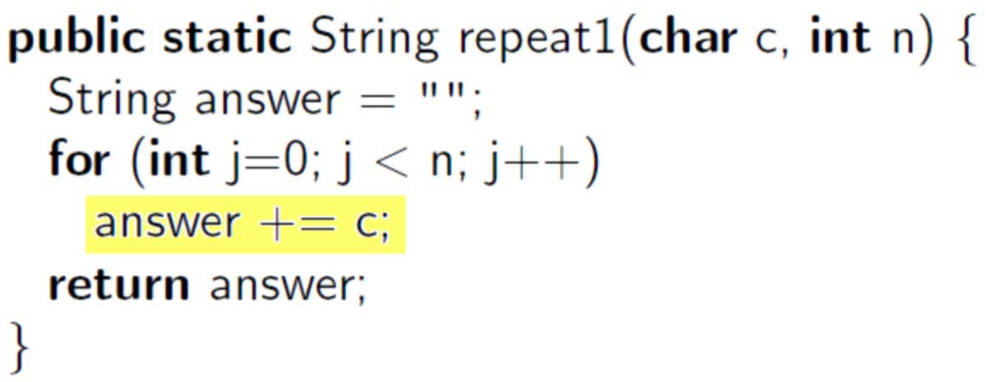
- Find the max of an array
  - Variable initializations and return are constants
  - Loop runs n-1 times
  - Comparison and assignment in the loop are constants

  - Running time of arrayMax: $c' \cdot (n-1) + c'' \Rightarrow O(n)$

Examples of Algorithm Analysis

- How many times currentMax is updated in arrayMax
  - Assume that data is randomly distributed (uniform distribution)
  - The probability that data[j] is the largest in data[0..j] is 1 / j
  - The expected number of times currentMax is updated is $H_n = 1 + 1/2 + 1/3 + \ldots + 1/n$
  - H_n is known as the n^{th} harmonic and is $O(\log n)$

Examples of Algorithm Analysis

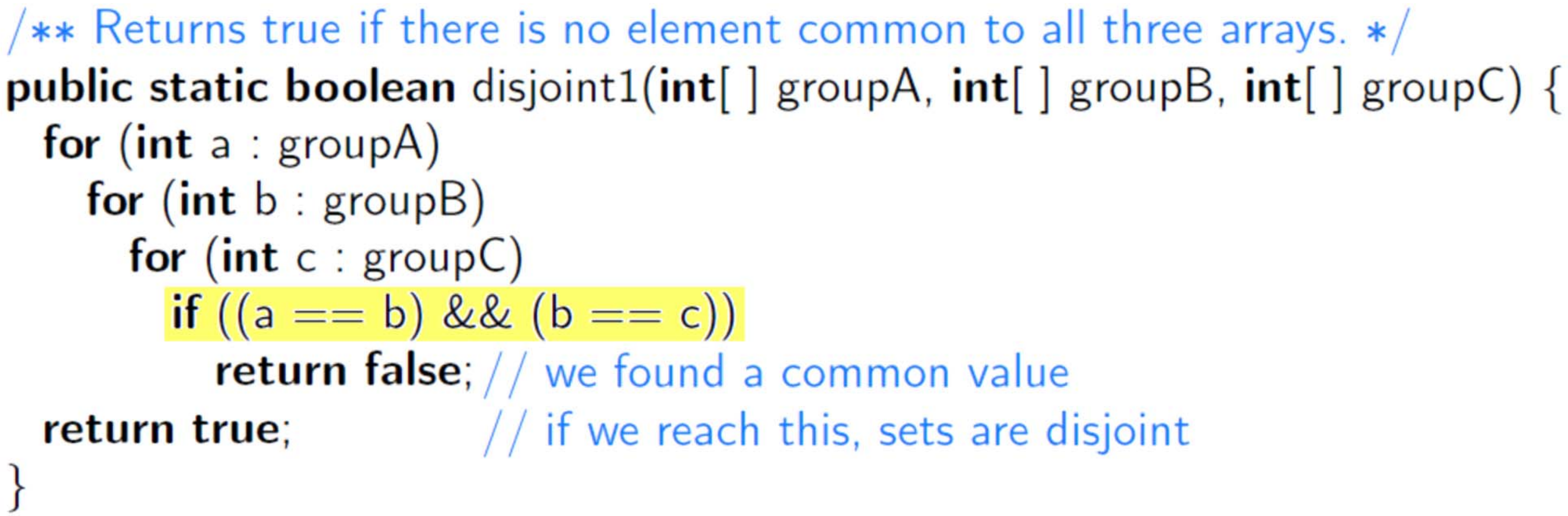


```
public static String repeat1(char c, int n) {
  String answer = "";
  for (int j=0; j < n; j++)
    answer += c;
  return answer;
}
```

- Strings in Java are immutable
  - To concatenate, a buffer for answer and c is allocated and answer and c are copied to the new buffer
  - Overall time taken for the concatenation is

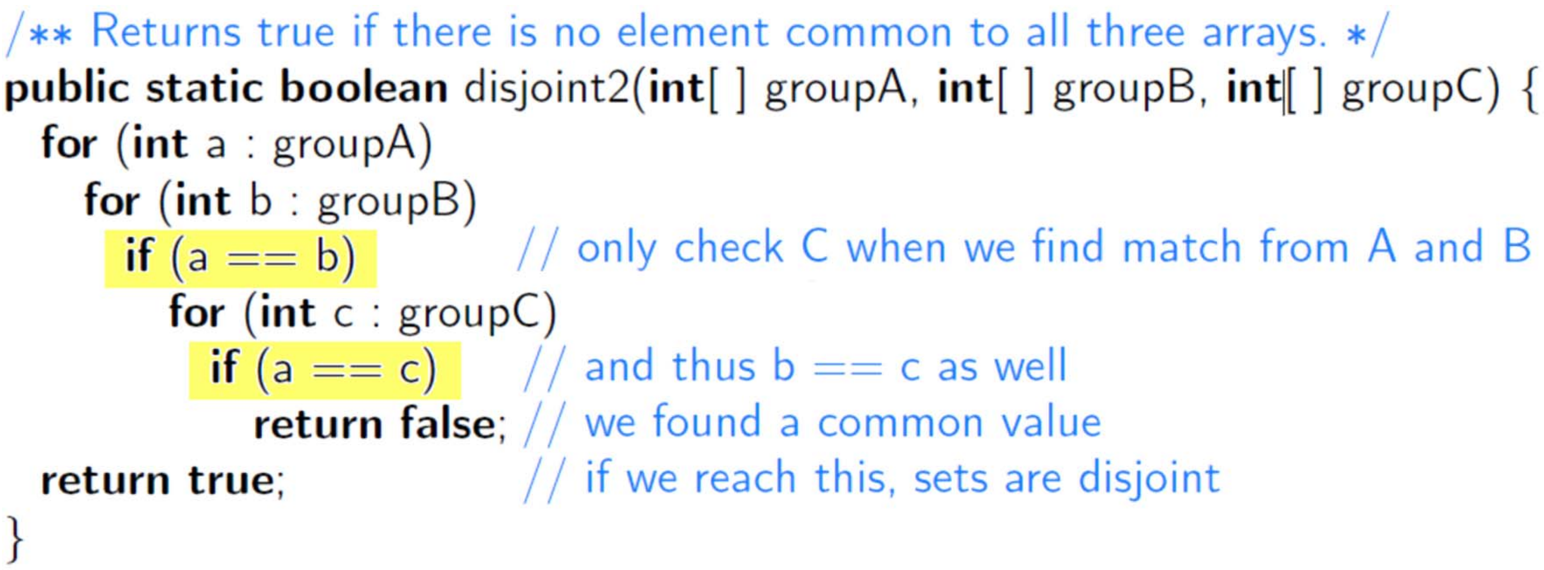
    1 + 2 + … + n
  - Hence the running time of repeat1 is $O(n^2)$

Examples of Algorithm Analysis



```
/** Returns true if there is no element common to all three arrays. */
public static boolean disjoint1(int[ ] groupA, int[ ] groupB, int[ ] groupC) {
  for (int a : groupA)
    for (int b : groupB)
      for (int c : groupC)
        if ((a == b) && (b == c))
          return false;   // we found a common value
  return true;            // if we reach this, sets are disjoint
}
```

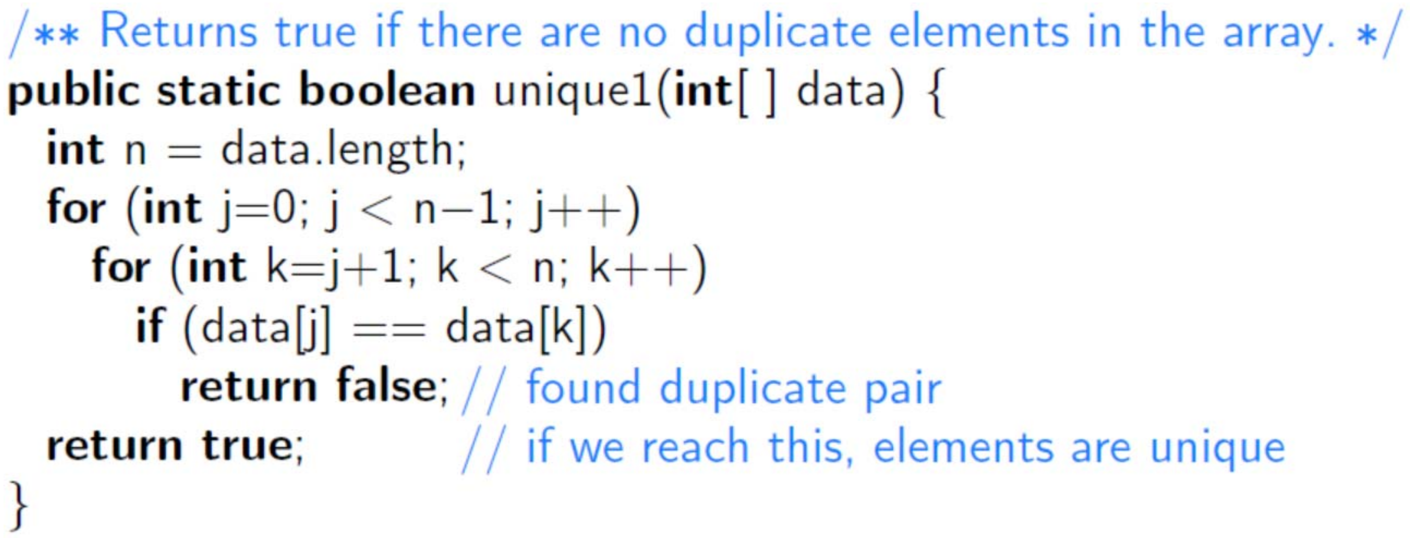
- Three-way set disjoints
  - Assume that *A*, *B*, and *C* don't have duplicate elements
  - Check if there is no such x as $x \in A$, $x \in B$, and $x \in C$
  - If $|A| = |B| = |C| = n$, the running time of disjoint1 is $O(n^3)$



```
/** Returns true if there is no element common to all three arrays. */
public static boolean disjoint2(int[ ] groupA, int[ ] groupB, int[ ] groupC) {
  for (int a : groupA)
    for (int b : groupB)
      if (a == b)            // only check C when we find match from A and B
        for (int c : groupC)
          if (a == c)        // and thus b == c as well
            return false;    // we found a common value
  return true;               // if we reach this, sets are disjoint
}
```

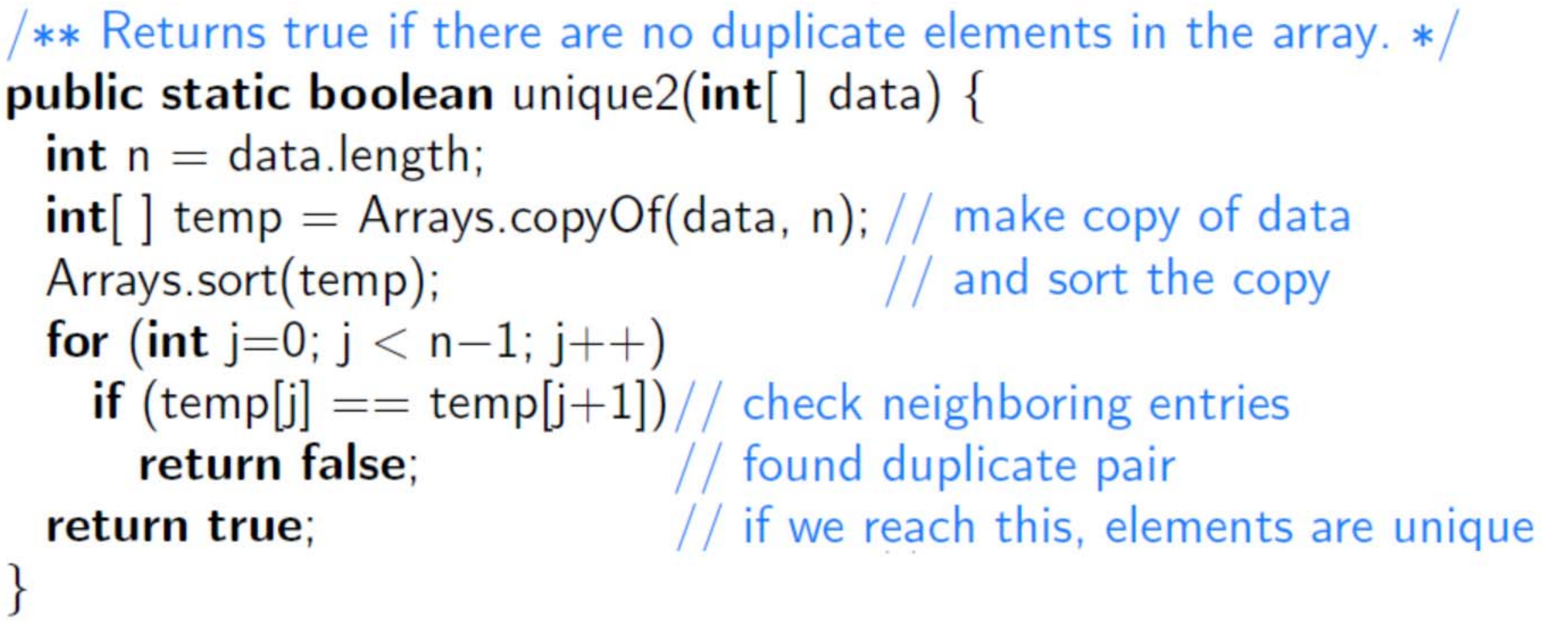
- Improved disjoint1: skip checking c when $a \neq b$
  - There are n^2 pairs of (a, b) to consider, but there are at most n pairs (a, b) such that a == b
    - Hence, a == c is checked at most n^2 times
  - a == b is checked n^2 times
  - The running time of disjoint2 is $O(n^2)$

Examples of Algorithm Analysis



```
/** Returns true if there are no duplicate elements in the array. */
public static boolean unique1(int[ ] data) {
  int n = data.length;
  for (int j=0; j < n−1; j++)
    for (int k=j+1; k < n; k++)
      if (data[j] == data[k])
        return false;        // found duplicate pair
  return true;               // if we reach this, elements are unique
}
```

- Check if there are no duplicate elements
  - The number of times data[j] == data[k] is checked $(n-1) + (n-2) + \ldots + 2 + 1$
  - The running time of unique1 is $O(n^2)$



```
/** Returns true if there are no duplicate elements in the array. */
public static boolean unique2(int[ ] data) {
  int n = data.length;
  int[ ] temp = Arrays.copyOf(data, n); // make copy of data
  Arrays.sort(temp);                    // and sort the copy
  for (int j=0; j < n−1; j++)
    if (temp[j] == temp[j+1])// check neighboring entries
      return false;          // found duplicate pair
  return true;               // if we reach this, elements are unique
}
```

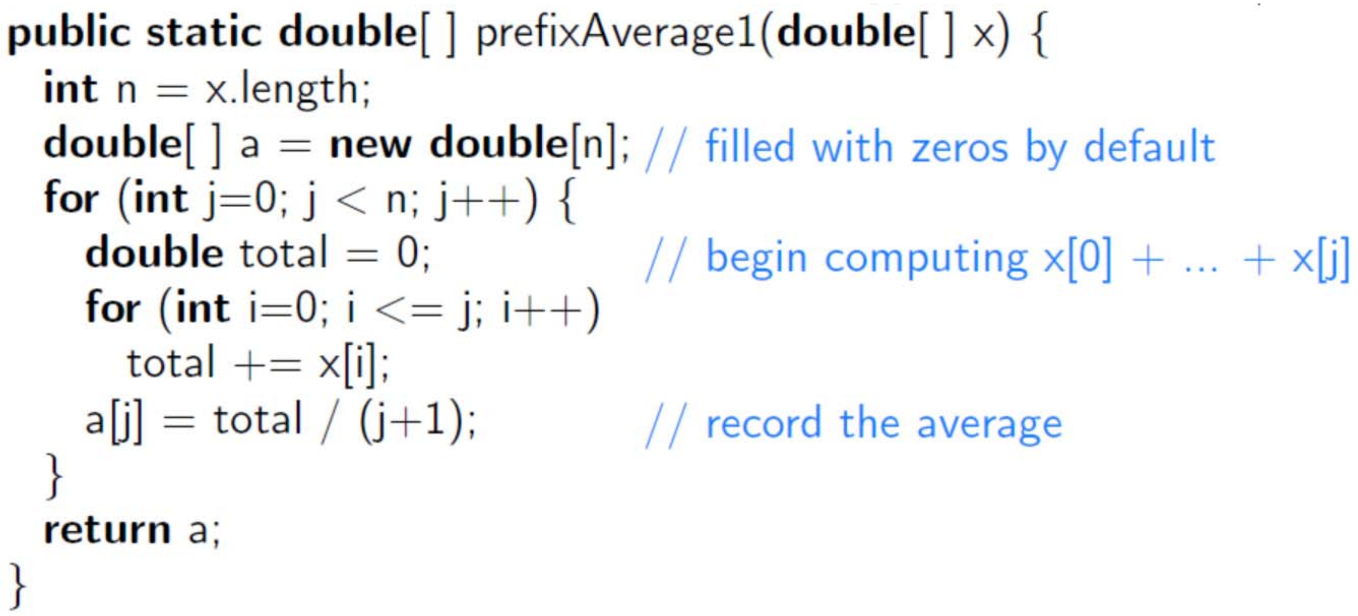
- Improve unique1 by sorting data
  - Arrays.copyOf takes O(*n*) time
  - Arrays.sort takes O(*n*·log *n*) time
  - temp[j]==temp[j+1] runs *n*-1 times
  - Hence, unique2 is O(*n*·log *n*)

Examples of Algorithm Analysis

- Prefix average
  - Given an array $x_0, \ldots, x_j$, compute a sequence a_j for $j = 0, \ldots, n-1$ such that

$$a_j = \frac{\sum_{i=0}^{j} x_i}{j+1}$$

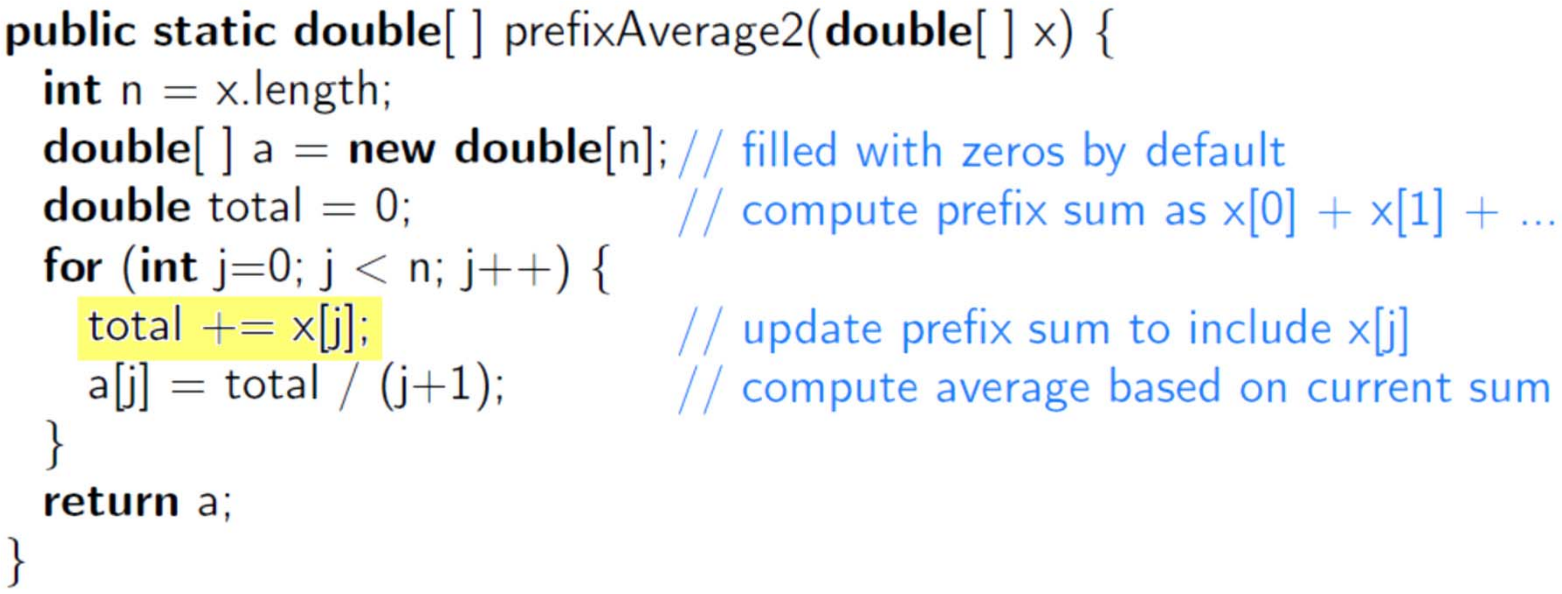
  - O(n^2) algorithm



```
public static double[ ] prefixAverage1(double[ ] x) {
  int n = x.length;
  double[ ] a = new double[n]; // filled with zeros by default
  for (int j=0; j < n; j++) {
    double total = 0;             // begin computing x[0] + ... + x[j]
    for (int i=0; i <= j; i++)
      total += x[i];
    a[j] = total / (j+1);         // record the average
  }
  return a;
}
```

Examples of Algorithm Analysis

- Prefix average
  - O(*n*) algorithm
    - Reuse total computed from the previous round



```
public static double[ ] prefixAverage2(double[ ] x) {
  int n = x.length;
  double[ ] a = new double[n]; // filled with zeros by default
  double total = 0;            // compute prefix sum as x[0] + x[1] + ...
  for (int j=0; j < n; j++) {
    total += x[j];             // update prefix sum to include x[j]
    a[j] = total / (j+1);      // compute average based on current sum
  }
  return a;
}
```