CSCI 136
Data Structures &
Advanced Programming

Bill Jannen

Lecture 7

Feb 19, 2017

Announcements

- Questions about Lab 2?
- Lab 3 will be handed out today
  - Lots of thinking…little typing
  - Problems can be done in any order!
  - Recursion can be frustrating…be patient!

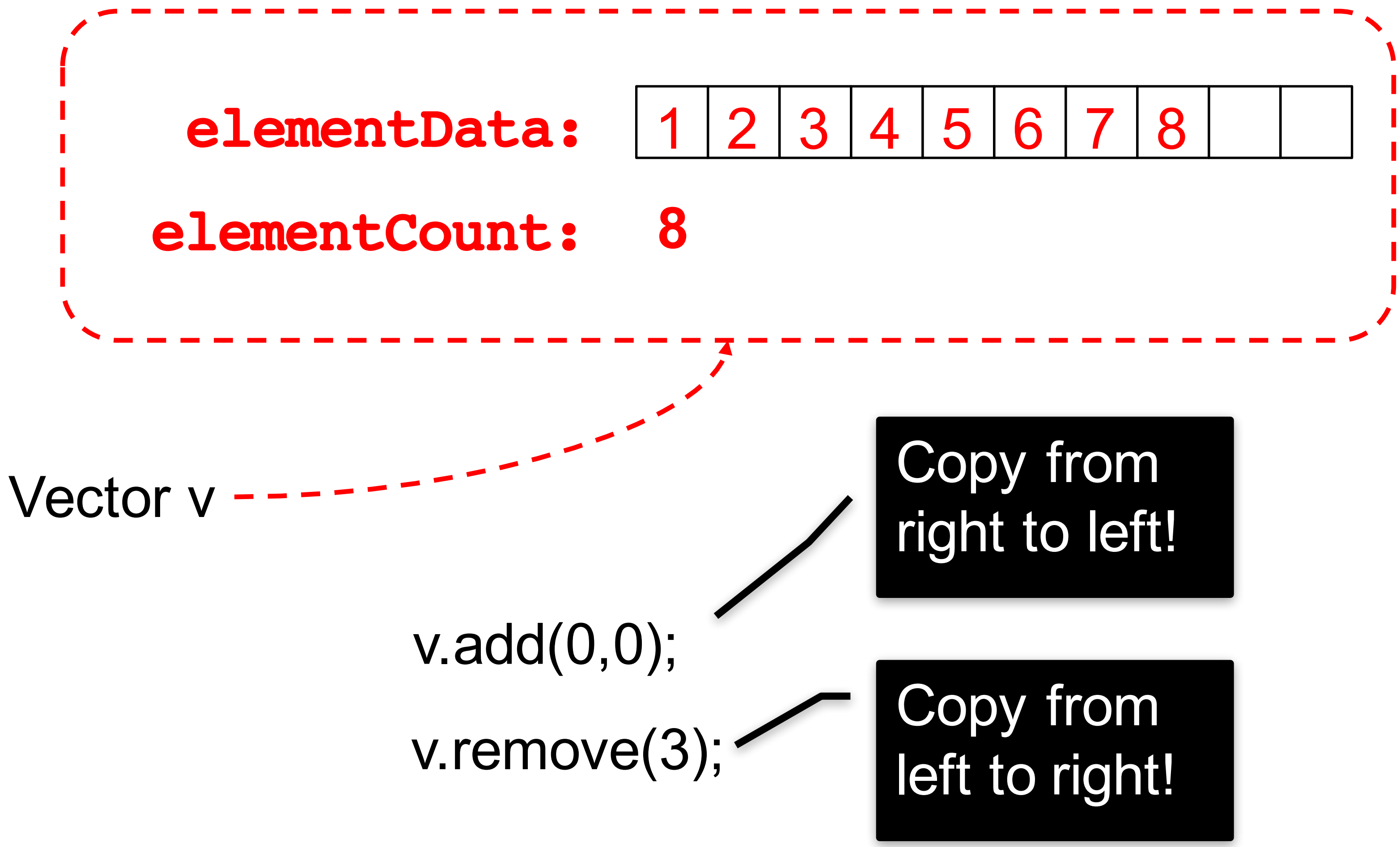
Last Time

- Generics
- Started implementing vectors
  - add(int index, E element), remove(int index):
    - Require *shifting*
  - add(int index, E element), add(E element):
    - Vectors must *grow* as we add more elements
    - How to expand a Vector's internal array?

Today's Outline

- Wrap up Vectors
- Learn about Big-O analysis
- Briefly discuss recursion
  - More next class, but quick recursion review for lab this week…

add(), remove()



elementData:
1 2 3 4 5 6 7 8
elementCount: 8
Vector v
v.add(0,0);
v.remove(3);
Copy from right to left!
Copy from left to right!

Growing Vectors

```
Vector v = new Vector(1);
v.add(0);
for (int i = 0; i < n; i++) {
        v.add(i);
}
```

Grow to meet new capacity.

Double the current capacity.

Growing Vectors

- Two ways to grow when adding n new elements to Vector:
  - Additive increase (add some constant factor)
    - Requires ~$n^2/2$ operations (or copies)
  - Multiplicative increase (double)
    - Requires ~n operations
  - Which is better?
  - Is there a tradeoff?

Vectors

- These questions relate to the time and space tradeoff
  - We could just as easily avoid all copy operations by making a huge Vector/array initially…
  - …but this wastes space and is inefficient

Shrinking the Array

- When should we shrink the array in Vector implementation?
  - When 1/2 full?
  - When 1/4 full?
- We shrink when 1/4 full…
- Can get bad performance if array size changes too frequently

Vector Constructors

```
protected Object elementData[]; // the data
protected int elementCount; // number of elements in vector


public Vector() {
   this(10);
}

public Vector(int initialCapacity) {
   elementData = new Object[initialCapacity];
   elementCount = 0;

}
```

Vector Constructors

```
protected Object elementData[]; // the data
protected int elementCount; // number of elements in vector
protected int capacityIncrement; // the rate of growth for vector

public Vector() {
   this(10);
}

public Vector(int initialCapacity) {
   elementData = new Object[initialCapacity];
   elementCount = 0;
   capacityIncrement = 0;
}

// pre: initialCapacity >= 0, capacityIncr >= 0
// post: constructs an empty vector with initialCapacity capacity
// that extends capacity by capacityIncr, or doubles if 0
public Vector(int initialCapacity, int capacityIncr) {
   elementData = new Object[initialCapacity];
   elementCount = 0;
   capacityIncrement = capacityIncr;
}
```

ensureCapacity()

```
public void add(E element) {
   ensureCapacity(elementCount+1);
   . . .
}


public void ensureCapacity(int minCapacity) {
   if (elementData.length < minCapacity) {
      if (capacityIncrement == 0) {
         //double the array
      } else {
         //grow by capacityIncrement
      }
      //copy elements to new array
   }
}
```

Growing the Array

- Vector.java
  - `ensureCapacity()`
- Chapter 3 of Bailey

Observations about Vectors

- How long does it take to add an element?
  - Varies – sometimes takes a lot longer if we have to grow array before adding element
- How long does it take to insert/remove an element in the middle of the Vector?
  - Might take a long time if we have to move several other elements
- Key insight: The running time depends on the size of the Vector!

Running Time Analysis

- We want *general tools* for understanding how running time and memory usage changes as the amount of data increases
- Example:
  - If I double my Vector's size, how much longer will it take to:
    - Find an element?
    - Insert an element at the front?
    - Remove an element from the middle?
    - Etc.

Measuring Computational Cost

- How can we measure the cost of a computation?
  - Absolute clock time
    - Problems?
      - Different machines have different clocks
      - Lots of other stuff happening (network, OS, etc)
      - Not consistent. Need lots of tests to predict future behavior

Measuring Computational Cost

- How can we measure the cost of a computation?
  - Count how many “expensive” operations were performed (i.e., array copies in Vector)
  - Count number of times “x” happens
    - For a specific event or action “x”
    - i.e., How many times a certain variable changes
  - Problems?
    - 64 vs 65? 100 vs 105? Does it really matter??

Measuring Computational Costs

- Rather than keeping exact counts, we want to know the *order of magnitude* of occurrences
  - 60 vs 600 vs 6000, not 65 vs 68
- We want to make comparisons without looking at details and without running tests
- Avoid using specific numbers or values
- Look for overall trends

Looking for Trends

- Rule of thumb: ignore constants (most of the time…)
- Examples:
  - Treat n and n/2 as same order of magnitude
  - $n^2/1000$, $2n^2$, and $1000n^2$ are "pretty much" just n^2 (behave in same way)
  - $a_0n^k + a_1n^{k-1} + a_2n^{k-2} + \cdots a_k$ is roughly n^k
- The key is to find the most significant or dominant term

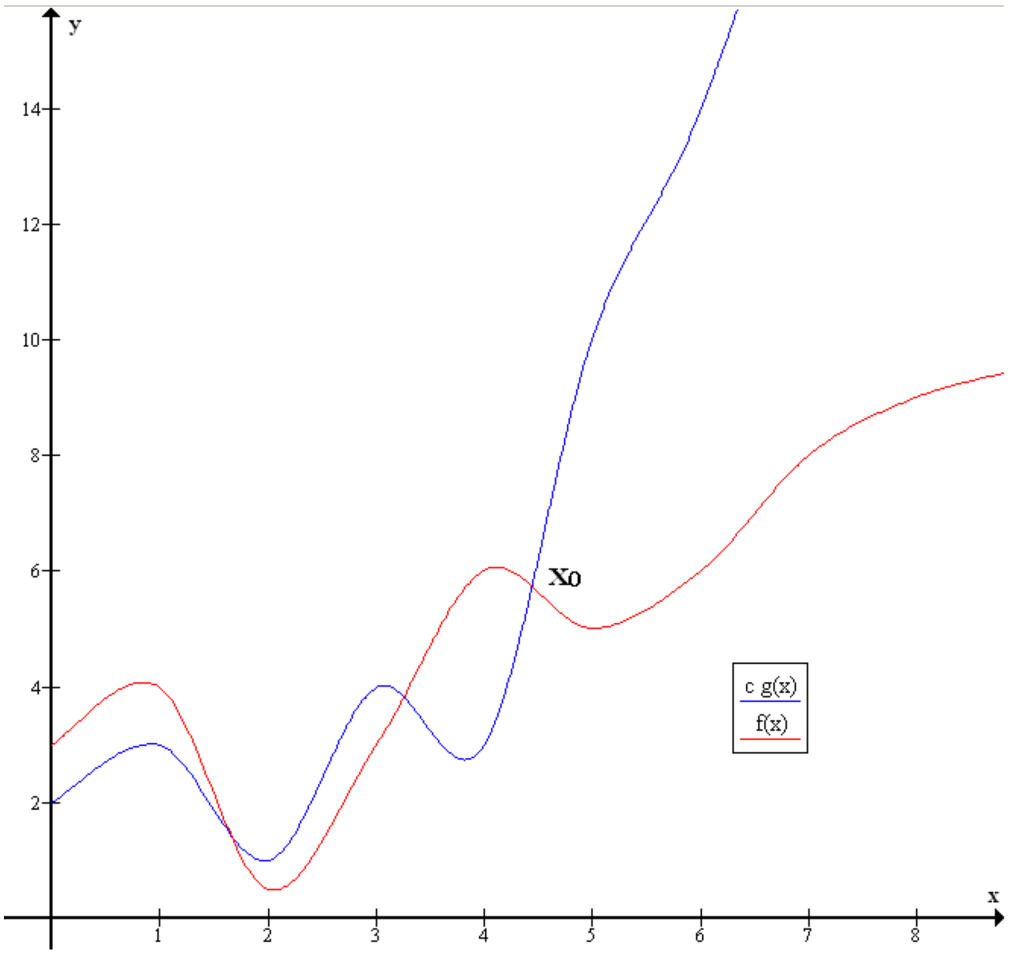
Asymptotic Bounds (Big-O Analysis)

- A function f(n) is O(g(n)) if and only if there exists positive constants c and n_0 such that

$$|f(n)| \leq c * g(n) \text{ for all } n \geq n_0$$

- "g" is bigger than "f" **for large n**
- Example:
  - $f(n) = n^2/2$ is $O(n^2)$
  - $f(n) = 1000n^3$ is $O(n^3)$
  - $f(n) = n/2$ is $O(n)$

$|f(n)| \leq c * g(n)$ for all $n \geq n_0$



[https://upload.wikimedia.org/wikipedia/commons/8/89/Big-O-notation.png]

Determining Upper Bound

- We usually want the smallest upper bound to estimate running time
- Example:
  - $f(n) = 3n^2$
  - $f(n)$ is $O(n^2)$
  - $f(n)$ is $O(n^3)$
  - $f(n)$ is $O(2^n)$
- Best estimate of running time is $O(n^2)$
- We might care about c and n_0 **in practice**, but focus on size of g when designing structures

Vector Operations

- For Object o, int i, and n elements:
  - set(i, o)
  - add(o)
  - add(i, o)
  - remove(i)
  - add(o) executed n times
  - add(i, o) executed n times

Vector Operations

- For Object o, int i, and n elements:
  - set(i, o) – O(1)
  - add(o) – O(1)
  - add(i, o) – O(n)
  - remove(i) – O(n)
  - add(o) executed n times – O(n)
  - add(i, o) executed n times – O(n^2)

Common Functions

For n = number of elements:

- O(1): constant time and space
- O(log n): divide and conquer algorithms, binary search
- O(n): linear dependence, simple list lookup
- O(n log n): divide and conquer sorting algorithms
- $O(n^2)$: matrix addition, selection sort
- $O(n^3)$: matrix multiplication
- $O(n^k)$: cell phone switching algorithms
- $O(2^n)$: color graph with 3 colors, satisfiability
- O(n!): traveling salesman problem

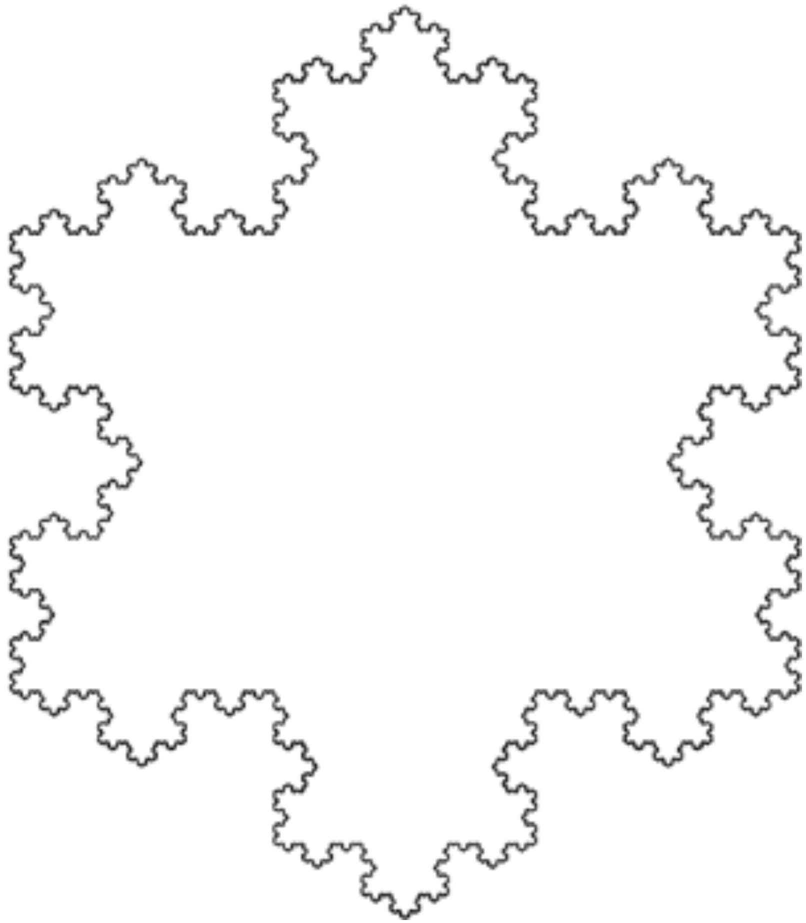
Input-dependent Running Times

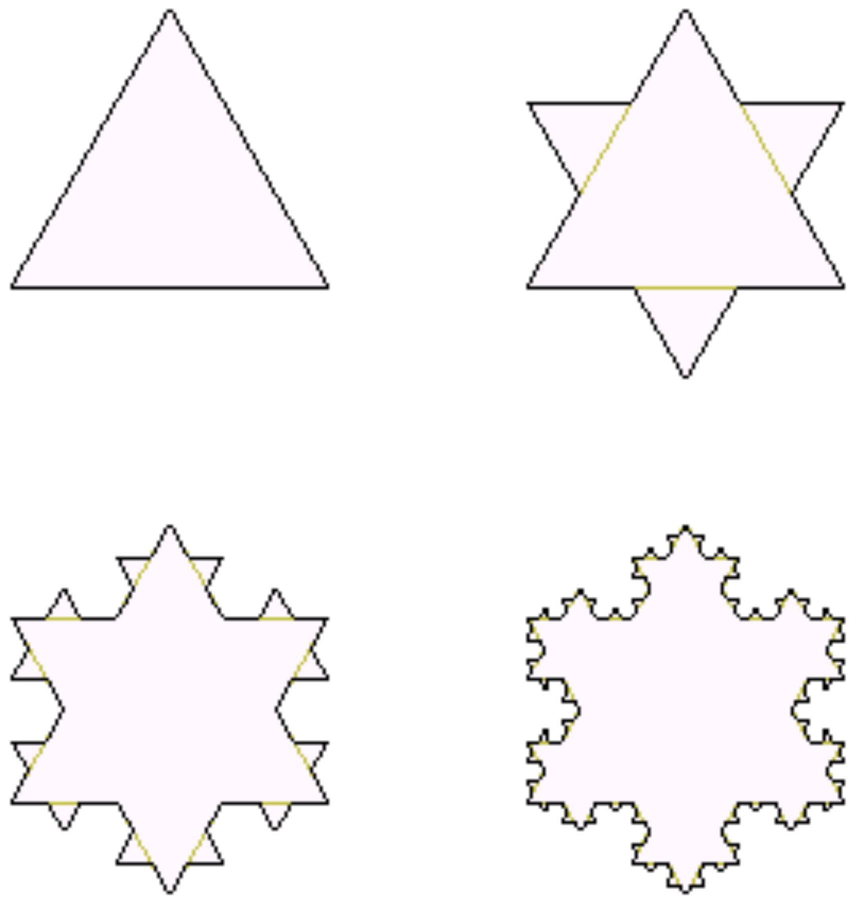
- Algorithms may have different running times for different input values
- Best case
  - Sort already sorted array in O(n)
  - Find item in first place that we look O(1)
- Worst case
  - Don't find item in list O(n)
  - Reverse order sort $O(n^2)$
- Average case
  - Linear search O(n)
  - Sort random array O(n log n)

Moving on…

Recursion

- General problem solving strategy
  - Break problem into smaller pieces
  - Sub-problems may look a lot like original - may in fact by smaller versions of same problem
- Examples





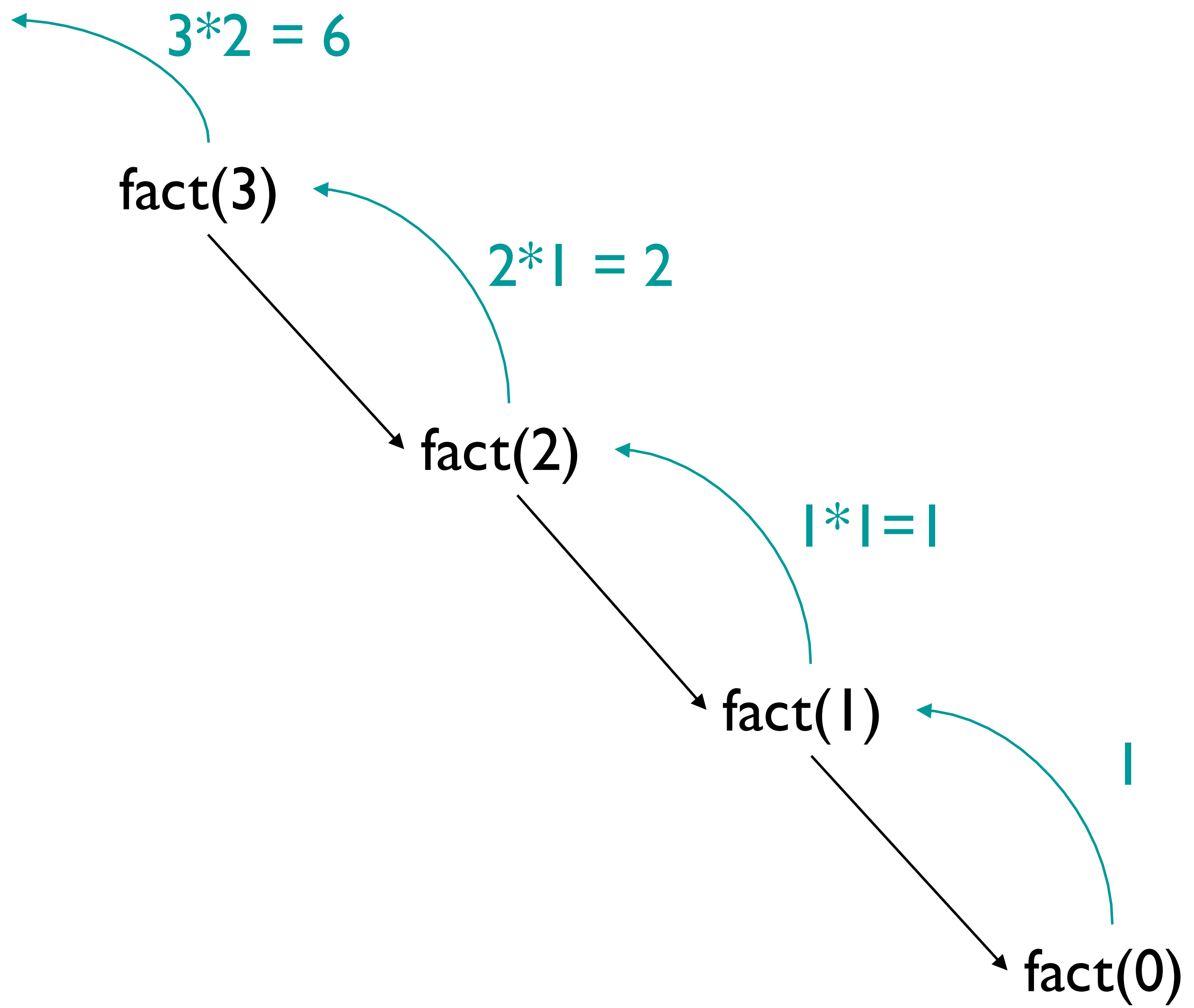
Recursion

- Many algorithms are recursive
  - Can be easier to understand (and prove correctness/state efficiency of) than iterative versions
- Today we will review recursion and Wednesday we will talk about techniques for reasoning about recursive algorithms

Factorial

- $n! = n \bullet (n-1) \bullet (n-2) \bullet \ldots \bullet 1$
- How can we implement this?
  - We could use a while loop…
- But we could also write it recursively
  - $n! = n \bullet (n-1)!$

Factorial



Factorial

- In recursion, we always use the same basic approach
- What's our base case?
  - n=0; fact(0) = 1
- What's our recursive case?
  - n>0; fact(n) = n • fact(n-1)

fact.java

```
public class fact{

    public static int fact(int n) {
       if (n==0) {
          return 1;
       }
       else {
          return n*fact(n-1);
       }
    }

    public static void main(String args[]) {
       System.out.println(fact(Integer.valueOf(args[0]).intValue()));
    }

}
```