10 A: Graph Algorithms III

CS1102S: Data Structures and Algorithms

Martin Henz

March 25, 2009

Generated on Tuesday 6th April, 2010, 14:21

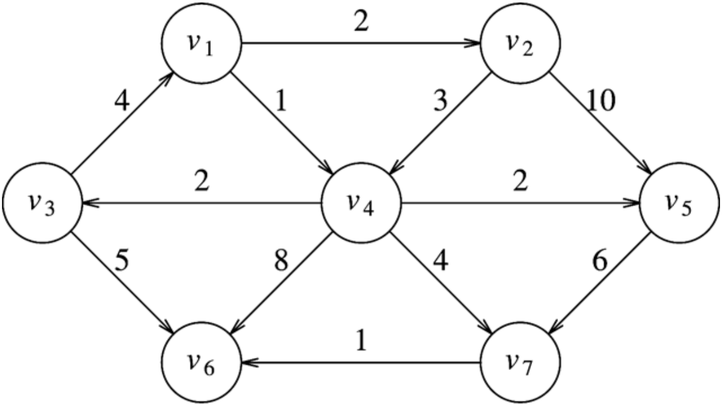
1. Correctness of Dijkstra’s Algorithm
   - Reviewing Dijkstra’s Algorithm
   - Correctness of Dijkstra’s Algorithm
   - Runtime Analysis
   - Negative Edge Costs
2. Maximum Flow and Minimum Spanning Tree
3. Puzzlers and Other Confusing Things

Single-Source Shortest-Path Problem

Problem

Given as input a weighted graph, $G = (V, E)$, and a distinguished vertex, s, find the shortest weighted path from s to every other vertex in G.

Example

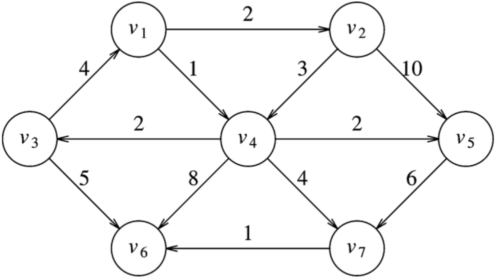


Shortest path from v_1 to v_6
has a cost of 6 and goes from v_1 to v_4 to v_7 to v_6.

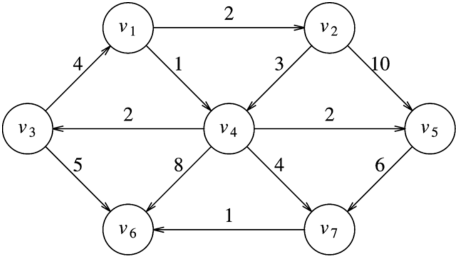
Dijkstra's Algorithm: Idea

Idea

Treat nodes in the order of shortest distance



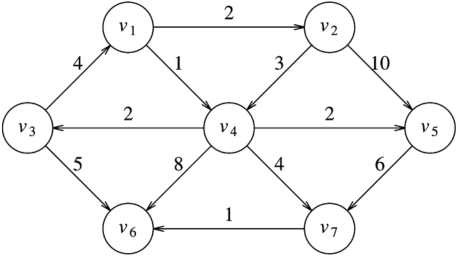
Example



Initial configuration:

v	*known*	d_v	p_v
v_1	F	0	0
v_2	F	∞	0
v_3	F	∞	0
v_4	F	∞	0
v_5	F	∞	0
v_6	F	∞	0
v_7	F	∞	0

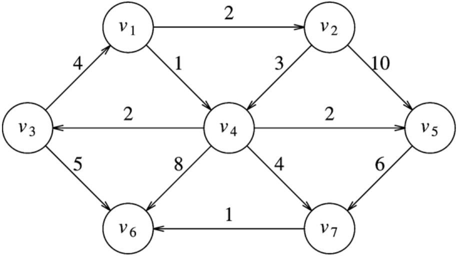
Example



After v_1 is declared known:

v	*known*	d_v	p_v
v_1	T	0	0
v_2	F	2	v_1
v_3	F	∞	0
v_4	F	1	v_1
v_5	F	∞	0
v_6	F	∞	0
v_7	F	∞	0

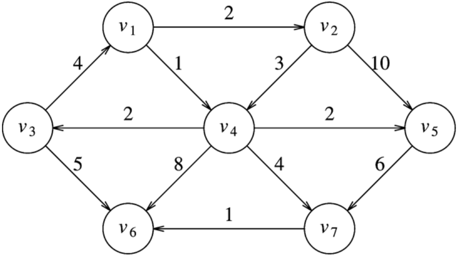
Example



After v_4 is declared known:

v	*known*	d_v	p_v
v_1	T	0	0
v_2	F	2	v_1
v_3	F	3	v_4
v_4	T	1	v_1
v_5	F	3	v_4
v_6	F	9	v_4
v_7	F	5	v_4

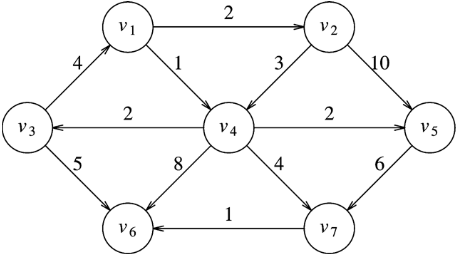
Example



After v_2 is declared known:

v	*known*	d_v	p_v
v_1	T	0	0
v_2	T	2	v_1
v_3	F	3	v_4
v_4	T	1	v_1
v_5	F	3	v_4
v_6	F	9	v_4
v_7	F	5	v_4

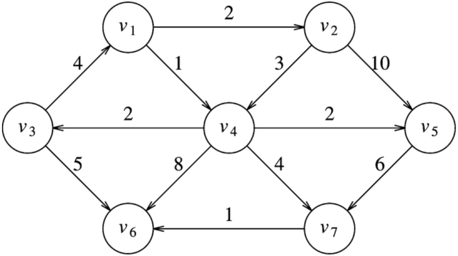
Example



After v_5 and then v_3 are declared known:

v	*known*	d_v	p_v
v_1	T	0	0
v_2	T	2	v_1
v_3	T	3	v_4
v_4	T	1	v_1
v_5	T	3	v_4
v_6	F	8	v_3
v_7	F	5	v_4

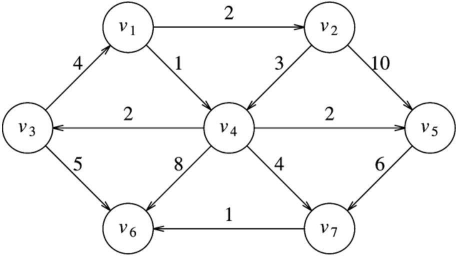
Example



After v_7 is declared known:

v	*known*	d_v	p_v
v_1	T	0	0
v_2	T	2	v_1
v_3	T	3	v_4
v_4	T	1	v_1
v_5	T	3	v_4
v_6	F	6	v_7
v_7	T	5	v_4

Example



After v_6 is declared known:

v	*known*	d_v	p_v
v_1	T	0	0
v_2	T	2	v_1
v_3	T	3	v_4
v_4	T	1	v_1
v_5	T	3	v_4
v_6	T	6	v_7
v_7	T	5	v_4

Pseudocode for Dijkstra's Algorithm

```
void dijkstra( Vertex s )
{
    for each Vertex v
    {
        v.dist = INFINITY;
        v.known = false;
    }

    s.dist = 0;

    for( ; ; )
    {
        Vertex v = smallest unknown distance vertex;
        if( v == NOT_A_VERTEX )
            break;
        v.known = true;

        for each Vertex w adjacent to v
            if( !w.known )
                if( v.dist + cvw < w.dist )
                {
                    // Update w
                    decrease( w.dist to v.dist + cvw );
                    w.path = v;
                }
    }
}
```

Shortest Subpath

Lemma

Any subpath of a shortest path must also be a shortest path.

Proof

By contradiction: Assume that a subpath $p = v_i \cdots v_j$ of the shortest path $q = v_1 \cdots p \cdots v_k$ is not a shortest path. Then there is a shorter path p' from v_i to v_j. Plug p' into q to get $q' = v_1 \cdots p' \cdots v_k$ shorter than q!

Definition of Shortest Distance

Notation

We use the notation

$$\delta(v, w)$$

to denote the length of the shortest path from v to w.

Distance

We call $\delta(v, w)$ the *distance* between v and w.

dist is a Relaxation

Lemma

At any point in time and for any vertex v, we have

$$v.dist \geq \delta(s, v)$$

Proof Idea

We show this by proving that whenever we set v.dist to a finite value, there exists a path of that length.

Order of Adding Vertices

Observation

Vertices are becoming known in order of increasing `dist` values.

Observation

Once a vertex becomes known, its `dist` value does not change.

Correctness of Dijkstra's Algorithm

```
void dijkstra( Vertex s )
{
    for each Vertex v
    {
        v.dist = INFINITY;
        v.known = false;
    }

    s.dist = 0;

    for( ; ; )
    {
        Vertex v = smallest unknown distance vertex;
        if( v == NOT_A_VERTEX )
            break;
        v.known = true;

        for each Vertex w adjacent to v
            if( !w.known )
                if( v.dist + cvw < w.dist )
                {
                    // Update w
                    decrease( w.dist to v.dist + cvw );
                    w.path = v;
                }
    }
}
```

Main Correctness Lemma

Lemma

When we set v.known = **true** then v.dist $= \delta(s, v)$.

Correctness of Dijkstra's algorithm

Each iteration through the outer loop makes one vertex known. At the end every vertex v is known and thus, according to the lemma, its dist value is $\delta(s, v)$.

Proving the Main Lemma

Proof by contradiction

Assume that there is a vertex v for which the claim does not hold.
Then, there must be a vertex u for which this is the case for the *first time* in the algorithm.
Using relaxation property, when we set u.known = true then u.dist $> \delta(s, u)$.

Situation

Analysis

Situation just before u.known = true: The value u.dist reflects the length of the path given by u.path.

The real shortest path

The real shortest path from s to u is shorter than u.dist. Consider the real shortest path $s \cdots u$.

Situation

Jumping into "unknown"

Since u.known still false, and s.known=true, there must be a pair of two neighboring vertices in the real shortest path such that r.known=true and t.known=false.

First pair

There must be a first pair x and y where this is the case.

Analysis of x and y

Processing of x

We have processed x, but not yet y. Since y's dist value is decreased by decrease(...), we know that

$$y\,.\,dist \le x\,.\,dist + c_{x,y}$$

Using hypothesis

Since x becomes known earlier than u, we have

$$x\,.\,dist = \delta(s, x)$$

Shortest subpath

$s \cdots x$ and $s \cdots xy$ is subpath of shortest path $\cdots u$:

$$\delta(s, y) = \delta(s, x) + c_{x,y} = x\,.\,dist + c_{x,y}$$

Recap

We have

$$y.dist \leq x.dist + c_{x,y}$$

and

$$\delta(s, y) = \delta(s, x) + c_{x,y} = x.dist + c_{x,y}$$

and thus: $y.\text{dist} \leq \delta(s, y)$
and therefore $y.\text{dist} = \delta(s, y)$

Finale

Since $y.\text{dist} = \delta(s, y)$, we have $y \neq u$.

Edge costs between y and u are non-negative, thus

$$\delta(s, y) \leq \delta(s, u)$$

and thus

$$y.dist = \delta(s, y) \leq \delta(s, u) < u.dist$$

If $y.\text{dist} < u.\text{dist}$, and since vertices become known in order of increasing distance, y would have become known before u, which contradicts the assumption that u is next vertex to become known!

Runtime Analysis of Dijkstra's Algorithm

```
void dijkstra( Vertex s )
{
    for each Vertex v
    {
        v.dist = INFINITY;
        v.known = false;
    }

    s.dist = 0;

    for( ; ; )
    {
        Vertex v = smallest unknown distance vertex;
        if( v == NOT_A_VERTEX )
            break;
        v.known = true;

        for each Vertex w adjacent to v
            if( !w.known )
                if( v.dist + cvw < w.dist )
                {
                    // Update w
                    decrease( w.dist to v.dist + cvw );
                    w.path = v;
                }
    }
}
```

Runtime Analysis

Naive implementation

Scan all vertices sequentially to find unknown vertex with minimum d_v: $O(|V|)$. Thus overall: $O(|V|^2)$

Runtime Analysis

Priority queue for unknown vertices

- `decrease(w.dist to v.dist + cvw);`

  uses insert

  There will be multiple entries in the priority queue for the same vertex!

- `Vertex v = smallest distance unknown vertex;`

  uses deleteMin and checks if the vertex is known already

Runtime Analysis

Priority queue for unknown vertices

```
decrease(w.dist to v.dist + cvw);
```

uses insert

There will be at most as many calls to decrease (and thus insert) as edges in the graph

Getting next vertex

```
Vertex v = smallest distance unknown vertex;
```

uses deleteMin and checks if the vertex is known already

There will be at most $|E|$ calls to deleteMin

Runtime Analysis

Overall Runtime

$O(|E| \log |V|)$.

Negative Edge Costs

Idea

Algorithm needs to change its mind; forget "known" vertices!

Keep improving cost

Add vertices to queue as long as cost improves

Shortest Path with Negative Edge Costs

```
void weightedNegative( Vertex s )
{
    Queue<Vertex> q = new Queue<Vertex>( );

    for each Vertex v
        v.dist = INFINITY;

    s.dist = 0;
    q.enqueue( s );

    while( !q.isEmpty( ) )
    {
        Vertex v = q.dequeue( );

        for each Vertex w adjacent to v
            if( v.dist + cvw < w.dist )
            {
                // Update w
                w.dist = v.dist + cvw;
                w.path = v;
                if( w is not already in q )
                    q.enqueue( w );
            }
    }
```

1. Correctness of Dijkstra's Algorithm

2. Maximum Flow and Minimum Spanning Tree
   - Maximum Flow
   - Minimum Spanning Tree

3. Puzzlers and Other Confusing Things

Maximum Flow

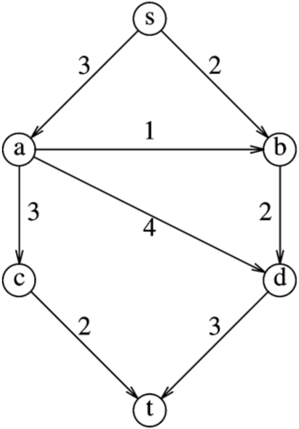
Interpretation of weights

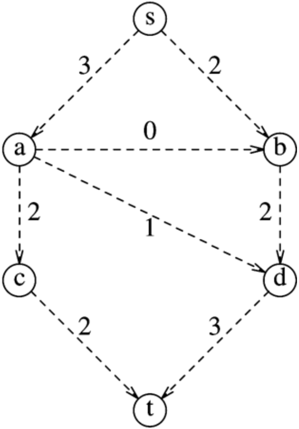
Every weight of an edge (v, w) represents a *capacity* of a flow passing from v to w.

Maximum Flow

Given two vertices s and t, compute the maximal flow achievable from s to t.

Example Graph and Maximum Flow





Idea

Identify an augmenting path

a path that has a feasible flow

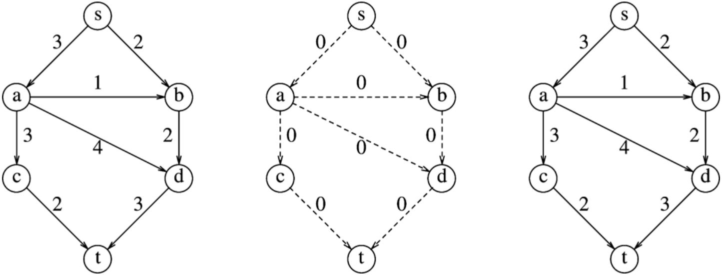
Add path to flow graph

keeping track of a flow that is already achieved

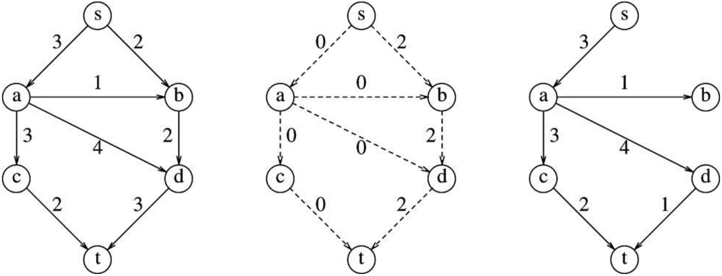
Remove path from residual graph

keeping track of the remaining flow that can still be exploited

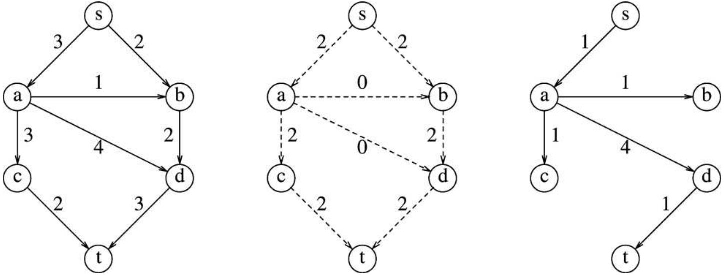
Example: Initial Setup



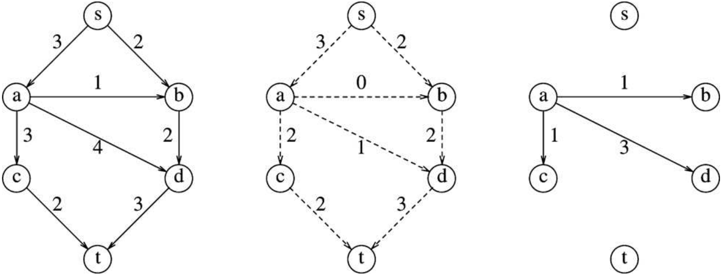
Example Run



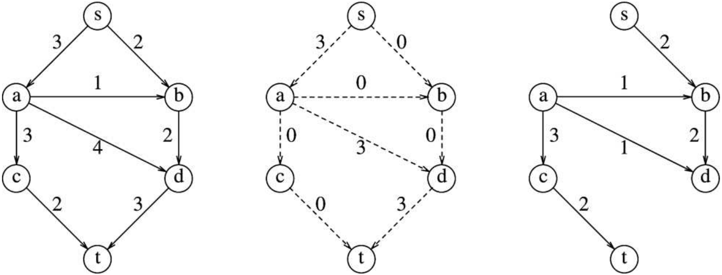
Example Run



Example Run



Counterexample: Suboptimal Solution



Idea

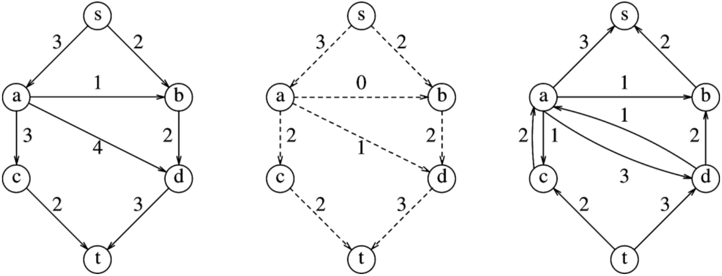
Allow algorithm to change its mind

Add reverse edge to residual graph, allowing a flow back in the opposite direction.

Consequence

This means that later, we can exploit an original flow which has been utilized already in the current flow graph, for a new augmenting path.

Example



Runtime Analysis

Assumption

Edge weights are integers

Each augmenting path makes "progress"

adding a flow of at least 1 to the flow graph.

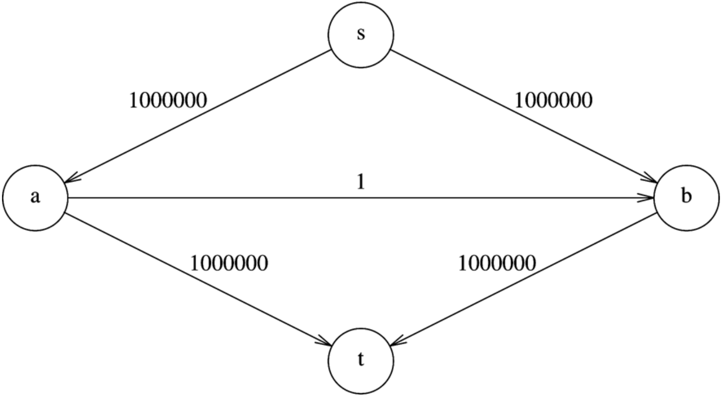
Finding augmenting paths

can be done in $O(N)$, using unweighted shortest path algorithm

Overall

$O(f \cdot |E|)$ where f is the maximum flow

Bad case



Smart Path Selection

Use Dijkstra's algorithm

for finding augmenting path (ignoring weights)

Number of augmentations

$O(|E| \log cap_{max})$, where cap_{max} is the maximum edge capacity

Overall runtime

$O(|E|^2 \log |V| \log cap_{max})$

Minimum Spanning Tree Problem

Input

Undirected weighted connected graph G

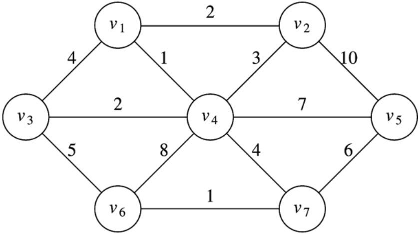
Spanning tree

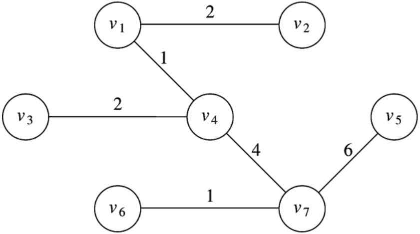
Tree that contains all of G’s vertices and only edges from G

Minimum spanning tree

Spanning tree whose sum of edge weights is minimal

An Example Graph and its MST





Algorithm Idea

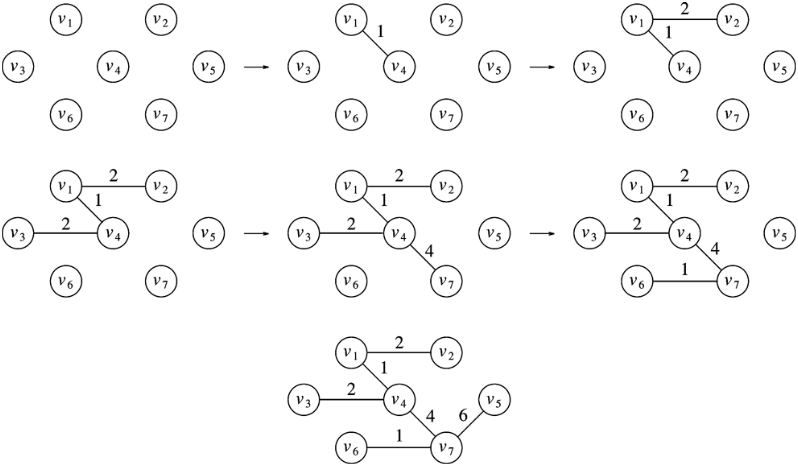
Similar to Dijkstra's Algorithm

Start "growing" the MST at arbitrary vertex. At each step, add an edge to MST with smallest weight.

Implementation

Keep edges from "known" to "unknown" vertices in a priority queue.

Example



Runtime Analysis

Without priority queue

$O(|V|^2)$

With priority queue

$O(|E| \log |V|)$

1. Correctness of Dijkstra's Algorithm
2. Maximum Flow and Minimum Spanning Tree
3. Puzzlers and Other Confusing Things
   - Values and References
   - Last Week's Puzzler: Printing Money
   - New Puzzler: Well, Dog My Cats!

Remember Lecture 2 A: Parameter Passing

Java uses pass-by-value parameter passing.

```
public static void tryChanging(int a) {
    a = 1;
    return;
}
...
int b = 2;
tryChanging(b);
System.out.println(b);
```

Remember Lecture 2 A: Parameter Passing with Objects

```
public static void tryChanging(SomeObject obj) {
    obj.someField = 1;
    obj = new SomeObject();
    obj.someField = 2;
    return;
}
...
SomeObject someObj = new SomeObject();
tryChanging(someObj);
System.out.println(someObj.someField);
```

Remember Lecture 7 A: Sorting

Input

Unsorted array of elements

Behavior

Rearrange elements of array such that the smallest appears first, followed by the second smallest etc, finally followed by the largest element

Will This Work?

```
public static <AnyType extends Comparable<? super A
  void mergeSort( AnyType [] a) {
    AnyType[] ret = ....; // declare helper array
    .... // here goes a program that places
    .... // the element of "a" into "ret" so
    .... // that "ret" is sorted
    a = ret;
    return;
}
...
Integer[] myArray = ...;
IterativeMergeSort.mergeSort(myArray);
```

Will This Work?

```
public static <AnyType extends Comparable<? super A
  void mergeSort( AnyType [] a) {
    AnyType[] ret = ....; // declare helper array
    .... // here goes a program that places
    .... // the element of "a" into "ret" so
    .... // that "ret" is sorted
    a = ret;
    return;
}
...
Integer[] myArray = ...;
IterativeMergeSort.mergeSort(myArray);
```

Answer: No! The assignment a = ret; has no effect on myArray!

Last Week's Puzzler: Printing Money

```
class Money {}
class Dollar extends Money {}
class MoneyPrinter {
  public void print(Money x) {
    System.out.println("Money!");
  }
}
class DollarPrinter extends MoneyPrinter {
  public void print(Dollar x) {
    System.out.println("Dollar!");
  }
}
```

Last Week's Puzzler: Printing Money

```
Money m = new Dollar();
MoneyPrinter mp = new DollarPrinter();
mp.print(m);
```

Last Week's Puzzler: Printing Money

```
Money m = new Dollar();
MoneyPrinter mp = new DollarPrinter();
mp.print(m);
```

What does mp.print(m) compile to?

Last Week's Puzzler: Printing Money

```
Money m = new Dollar();
MoneyPrinter mp = new DollarPrinter();
mp.print(m);
```

What does mp.print(m) compile to?

```
invokevirtual MoneyPrinter.print(Money;)V
```

Thus the JVM begins to search for a matching method at class MoneyPrinter

Output: "Money"

Last Week's Puzzler: Printing Money

```
Money m = new Dollar();
DollarPrinter dp = new DollarPrinter();
dp.print(m);
```

What does dp.print(m) compile to?

```
invokevirtual DollarPrinter.print(Money;)V
```

Thus the JVM begins to search for a matching method at class DollarPrinter, but finds the matching method in MoneyPrinter.

Output: "Money"

Last Week's Puzzler: Printing Money

```
Dollar d = new Dollar();
MoneyPrinter mp = new DollarPrinter();
mp.print(d);
```

What does mp.print(d) compile to?

```
invokevirtual MoneyPrinter.print(Money;)V
```

Note that the compiler changes the type of d to Money.
The JVM begins to search for a matching method at class MoneyPrinter
Output: “Money”

Last Week's Puzzler: Printing Money

```
Dollar d = new Dollar();
DollarPrinter mp = new DollarPrinter();
dp.print(d);
```

What does dp.print(d) compile to?

```
invokevirtual DollarPrinter.print(Dollar;)V
```

Thus the JVM begins to search for a matching method at class DollarPrinter and finds it there!

Output: "Dollar"

New Puzzler: Well, Dog My Cats!

```
class Counter {
    private static int count;
    public static void increment() { count++; }
    public static int getCount()   { return count; }
}
class Dog extends Counter {
    public Dog() {}
    public void woof() { increment(); }
}
class Cat extends Counter {
    public Cat() {}
    public void meow() { increment(); }
}
```

New Puzzler: Well, Dog My Cats!

```
public class Ruckus {
  public static void main(String[] args) {
    Dog dogs[] = { new Dog(), new Dog() };
    for (int i = 0; i < dogs.length; i++)
      dogs[i].woof();
    Cat cats[] = { new Cat(),new Cat(),new Cat() };
    for (int i = 0; i < cats.length; i++)
      cats[i].meow();
    System.out.print(Dog.getCount()+" woofs and ");
    System.out.println(Cat.getCount() + " meows");
  }
}
```

What is printed by this program?