A+ Computer Science

RECURSION

A+ Computer Science
www.apluscompsci.com

Recursion

Recursion occurs when a method calls itself.

Recursion

```
public class RecurOne
{
  public void run(int x)
  {
    out.println(x);
    run(x+1);
  }
  public static void main(String args[]  )
  {
    RecurOne test = new RecurOne();
    test.run(1);
  }
}
```

Will it stop?

OUTPUT

1
2
3
4
5
. . . . .
stack overflow

recursionone.java

Recursion

A recursive method must have a stop condition/ base case.

Recursive calls will continue until the stop condition is met.

Recursion

```
public class RecursionTwo
{
  public void run(int x )
  {
    out.println(x);
    if(x<5)
      run(x+1);
  }
  public static void main(String args[]  )
  {
    RecursionTwo test = new RecursionTwo();
    test.run(1);
  }
}
```

base case
It will stop!

OUTPUT

1
2
3
4
5

Recursion

```
public class RecursionThree
{
  public void run(int x )
  {
    if(x<5)
      run(x+1);
    out.println(x);
  }
  public static void main(String args[]  )
  {
    RecursionThree test = new RecursionThree ();
    test.run(1);
  }
}
```

base case

OUTPUT

5
4
3
2
1

recursiontwo.java
recursionthree.java

Recursion

```
class DoWhile
{
  public void run( )
  {
    int x=0;
    do{
      x++;
      out.println(x);
    }while(x<10);          //condition
  }
  public static void main(String args[]  )
  {
    DoWhile test = new DoWhile();
    test.run( );
  }
}
```

The Stack

When you call a method, an activation record for that method call is put on the stack with spots for all parameters/arguments being passed.

The Stack

AR1- method() call

The Stack

AR2- method() call
AR1- method() call

The Stack

AR3- method() call
AR2- method() call
AR1- method() call

The Stack

AR4- method() call
AR3- method() call
AR2- method() call
AR1- method() call

The Stack

AR3- method() call
AR2- method() call
AR1- method() call

The Stack

AR2- method() call
AR1- method() call

The Stack

As each call to the method completes, the instance of that method is removed from the stack.

AR1- method() call

Recursion

```
public class RecursionTwo
{
  public void run(int x )
  {
    out.println(x);
    if(x<5)
      run(x+1);
  }
  public static void main(String args[]  )
  {
    RecursionTwo test = new RecursionTwo();
    test.run(1);
  }
}
```

base case
It will stop!

OUTPUT

1
2
3
4
5

Recursion

```
public class RecursionThree
{
   public void run(int x )
   {
     if(x<5)
        run(x+1);
     out.println(x);
   }
   public static void main(String args[]  )
   {
     RecursionThree test = new RecursionThree();
     test.run(1);
   }
}
```

base case

OUTPUT

5
4
3
2
1

Why does this output differ from recur2?

Tracing Recursion

```
int fun(int y)
{
   if(y<=1)
      return 1;
   else
      return fun(y-2) + y;
}

//test code in client class
out.println(test.fun(5));
```

AR3
y
1   return 1

AR2
y
3   return AR3 + 3 4

AR1
y
5   return AR2 + 5

9

Tracing Recursion

```
int fun( int x, int y)
{
  if( y < 1)
     return x;
  else
     return fun( x, y - 2) + x;
}

//test code in client class
out.println(test.fun(4,3));
```

AR3

x y

4 -1 return 4

AR2

x y

4 1 return AR3 + 4

8

AR1

x y

4 3 return AR2 + 4

12

recursionfour.java
recursionfive.java

Tracing Recursion

```
int fun(int x, int y)
{
  if ( x == 0 )
    return x;
  else
    return x+fun(y-1,x);
}
```

OUTPUT

16

What would fun(4,4) return?

recursionsix.java

Split / Tail Recursion

```
public String recur(String s)
{
  int len = s.length();
  if(len>0)
      return recur(s.substring(0,len-1)) +
                                        s.charAt(len-1);
  return "";
}
```

Split / Tail Recursion

```
public String recur(String s)
{
  int len = s.length();
  if(len>0)
      return s.charAt(len-1) +
                          recur(s.substring(0,len-1));
  return "";
}
```

recursionseven.java
recursioneight.java

The Stack

call out.println(recur("abc"))

```
public String recur(String s)
{
  int len = s.length();
  if(len>0)
      return recur(s.substring(0,len-1)) +
                                         s.charAt(len-1);
  return "";
}
```

The Stack

call out.println(recur("abc"))

AR stands for activation record. An AR is placed on the stack every time a method is called.

AR1 – s="abc"
return AR2 + c

The Stack

AR2 – s="ab" return AR3 + b
AR1 – s="abc" return AR2 + c

The Stack

AR3 – s="a" return AR4 + a
AR2 – s="ab" return AR3 + b
AR1 – s="abc" return AR2 + c

The Stack

AR4 – s=""   **return ""**
AR3 – s="a"   **return AR4 + a**
AR2 – s="ab"   **return AR3 + b**
AR1 – s="abc"   **return AR2 + c**

The Stack

AR3 – s="a"   **return a**
AR2 – s="ab"   **return AR3 + b**
AR1 – s="abc"   **return AR2 + c**

The Stack

AR2 – s="ab" **return ab**
AR1 – s="abc" **return AR2 + c**

The Stack

call out.println(recur("abc"))

OUTPUT

abc

AR1 – s="abc"
return abc

What is the point?

If recursion is just a loop, why would you just not use a loop?

Recursion is a way to take a block of code and spawn copies of that block over and over again. This helps break a large problem down into smaller pieces.

Counting Spots

If checking 0 0, you would find 5 @s are connected.

```
@ - @ - - @ - @ @ @
@ @ @ - @ @ - @ - @
- - - - - - - @ @ @
- @ @ @ @ @ - @ - @
- @ - @ - @ - @ - @
@ @ @ @ @ @ - @ @ @
- @ - @ - @ - - - @
- @ @ @ - @ - - - -
- @ - @ - @ - @ @ @
- @ @ @ @ @ - @ @ @
```

@ at spot [0,0]
@ at spot [0,2]
@ at spot [1,0]
@ at spot [1,1]
@ at spot [1,2]

The exact same checks are made at each spot.

Counting Spots

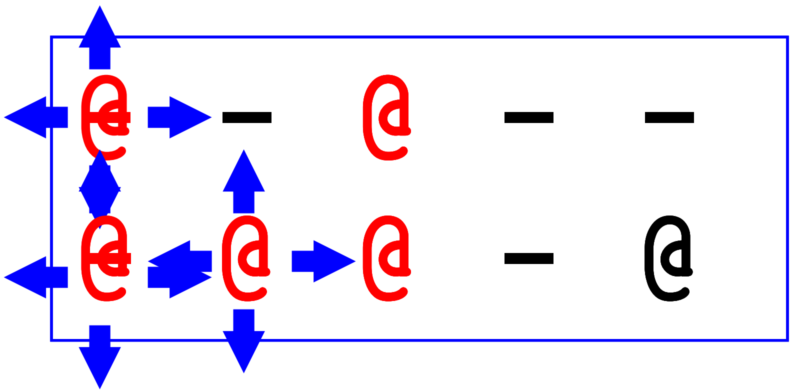
if (r and c are in bounds and
current spot is a @)
mark spot as visited
bump up current count by one
recur up
recur down
recur left
recur right

This same block of
code is recreated with each recursive call. The exact same code is used to check many different locations.

Counting blob problems are very common contest and technical interview problems.

Counting Spots

if (r and c are in bounds and
current spot is a @)
mark spot as visited
bump up current count by one
recur up
recur down
recur left
recur right



Sorting and Searching with Recursion

Search Algorithms

The Binary Search works best with sorted lists. The Binary search cuts the list in half each time it checks for for the specified value. If the value is not found, the search continue in the half most likely to contain the value.

Binary Search

```
int binarySearch (int [] stuff, int val )
{
  int bot= 0, top = stuff.length-1;
  while(bot<=top)
  {
    int middle = (bot + top) / 2;
    if (stuff[middle] == val)  return middle;
    else
      if (stuff[middle] > val)
        top = middle-1;
      else
        bot = middle+1;
  }
  return -1;
}
```

BinarySearch

```
public static int binarySearch (int [] stuff, int v,
                                            int b, int t )
{
  if(b<=t)
  {
    int m = (b + t) / 2;
    if (stuff[m] == v)
       return m;
    if (stuff[m] > v)
         return binarySearch(stuff, v, b, m-1);
    return binarySearch(stuff, v, m+1, t);
  }
  return -1;
}
```

BinarySearch

Search Algorithms

```
int[] stuff = {1,6,8,10,14,22,30,50};
```

0 + 7 = 7 / 2 = 3
stuff[3] = 10

4 + 7 = 11 div 2 = 5
stuff[5] = 22

6 + 7 = 13 div 2 = 6
stuff[6] = 30

If you are searching for 25, how many times will you check the stuff?

Search Algorithms

Given a list of N items.

What is the next largest power of 2?

If N is 100, the next largest power of 2 is 7.

$Log_2(100) = 6.64386$

$2^7 = 128$.

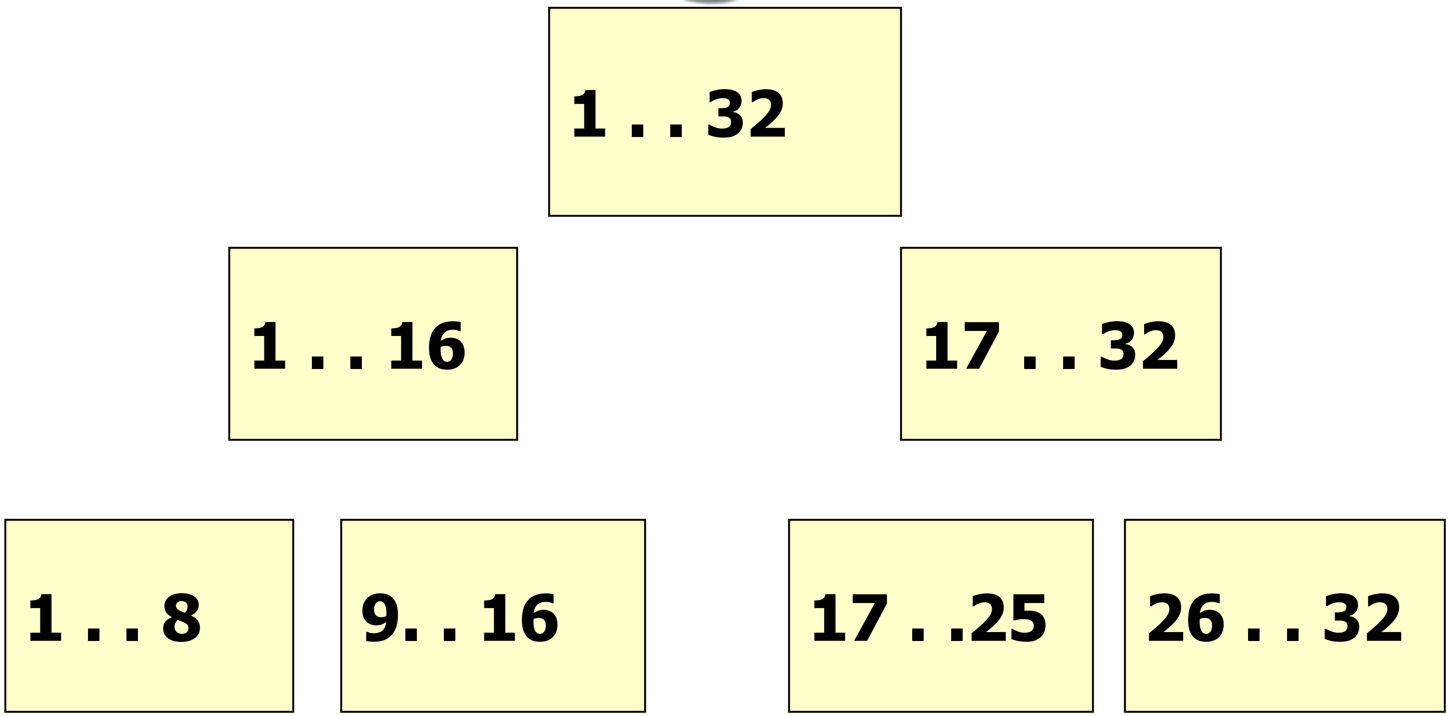
It would take 7 checks max to find if an item existed in a list of 100 items.

Sort Algorithms

Merge sort splits the list into smaller sections working its way down to groups of two or one. Once the smallest groups are reached, the merge method is called to organize the smaller lists. Merge copies from the sub list to a temp array. The items are put in the temp array in sorted order.

Merge Sort

Sort Algorithms



Merge sort chops in half repeatedly to avoid processing the whole list at once.

Sort Algorithms

```
void mergeSort(Comparable[] stuff, int front, int back)
{
  int mid = (front+back)/2;
  if(mid==front) return;
  mergeSort(stuff, front, mid);
  mergeSort(stuff, mid, back);
  merge(stuff, front, back);
}
```

Collections.sort( ) uses the mergeSort.

Arrays.sort( ) uses mergeSort for objects.

Merge W/Objects

```
public static void merge(
                    Comparable[] stuff, int front, int back) {
  int dif = back-front, spot = 0;
  Comparable[] temp = new Comparable[ dif ];
  int beg = front, mid = (front+back)/2, saveMid = mid;
  while( beg<saveMid && mid<back ) {
    if(stuff[ beg ].compareTo(stuff[ mid ])<0)
      temp[ spot++ ]= stuff[ beg++ ];
    else
      temp[ spot++ ]= stuff[ mid++ ];
  }
  while( beg < saveMid )
    temp[ spot++ ]= stuff[ beg++ ];
  while( mid < back )
    temp[ spot++ ]= stuff[ mid++ ];
  for(int i = 0; i < dif; ++i)
    stuff[front+i]=temp[i];
}
```

Sort Algorithms

Original List

Integer[] stuff = {90,40,20,30,10,67};

pass 0 - 90 20 40 30 67 10
pass 1 - 20 40 90 30 67 10
pass 2 - 20 40 90 30 10 67
pass 3 - 20 40 90 10 30 67
pass 4 - 10 20 30 40 67 90

Merge Sort

The mergeSort has a $N*Log_2N$ BigO.

mergeSort

The mergeSort method alone has a Log_2N run time, but cannot be run without the merge method.

Merge

The merge method alone has an N run time and can be run without the mergeSort method.

Runtime Analysis

```
for( int i=0; i<20; i++)
   System.out.println(i);
```

```
for( int j=0; j<20; j++)
   for( int k=0; k<20; k++)
      System.out.println(j*k);
```

Which section of code would execute the fastest?

Runtime Analysis

```
ArrayList<Integer> iRay;
iRay = new ArrayList<Integer>();
for( int i=0; i<20; i++)
   iRay.add(i);
```

```
ArrayList<Double> dRay;
dRay = new ArrayList<Double>();
for( int j=0; j<20; j++)
   dRay.add(0,j);
```

Which section of code would execute the fastest?

Work on Programs!

Crank Some Code!

A+ Computer Science

RECURSION