

# CSE 3318

Week of 06/12/2023

Instructor : Donna French

# Asymptotic Notation

The average running time of linear search grows as the array grows.

The notation used to describe this behavior is

$\Theta(n)$

Big Theta of  $n$

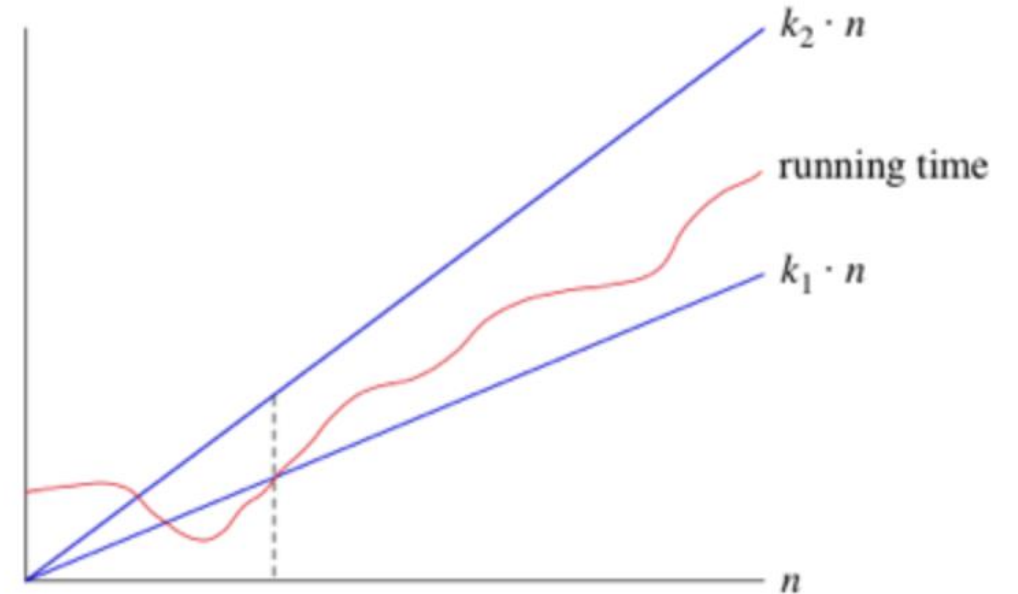
# Asymptotic Notation

$\Theta(n)$

When an algorithm's run time is described using  $\Theta$  notation, we are stating there is an

**asymptotically tight bound**

on the running time meaning that the run time will be tightly bound within a range once  $n$  gets big enough.



# Asymptotic Notation

$\Theta(n)$

Think of  $\Theta$  notation as a useful range between a narrow upper and lower bounds.

What is the temperature going to be today?

Well, it won't get hotter than  $120^{\circ}$  F or colder than  $-23^{\circ}$  F today – guaranteed!

A narrower (and more useful) range would be a high of  $94^{\circ}$ F and a low of  $74^{\circ}$ .

# Asymptotic Notation

If we have an algorithm that runs in constant time,

finding the smallest element of a sorted array

we would describe the run time as a function of  $n$ , which in  $\Theta$  notation would be

$$\Theta(n^0)$$

The algorithm's run time is within some constant factor of 1.

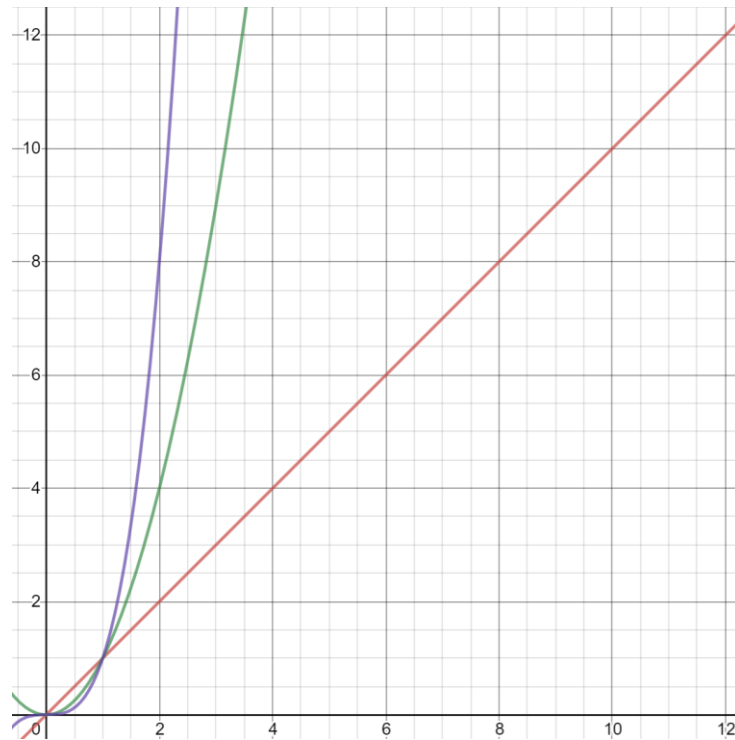
Because  $n^0 = 1$ , you will see this written as

$$\Theta(1)$$

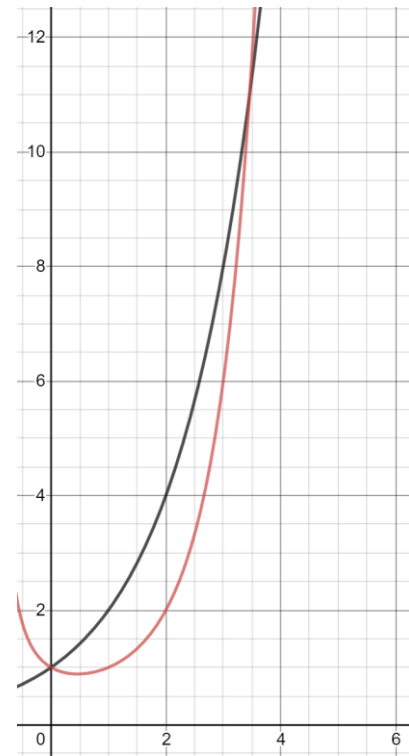
# Asymptotic Notation

Here's a list of functions in  $\Theta$  asymptotic notation from slowest to fastest in terms of growth

$\Theta(1)$   
 $\Theta(\log_2 n)$   
 $\Theta(n)$   
 $\Theta(n \log_2 n)$   
 $\Theta(n^2)$   
 $\Theta(n^2 \log_2 n)$   
 $\Theta(n^3)$   
 $\Theta(2^n)$   
 $\Theta(n!)$



$n$  and  $n^2$  and  $n^3$



$2^n$  and  $n!$



$\log_2 n$  and  $n \log_2 n$  and  $n^2 \log_2 n$

# The Role of Algorithms in Computing

Two algorithms for sorting : insertion sort and merge sort

## Insertion Sort

Takes  $c_1 n^2$  to sort  $n$  items

$c_1$  is a constant that does not depend on  $n$

## Merge Sort

Takes  $c_2 n \log_2 n$  to sort  $n$  items

$c_2$  is a constant that does not depend on  $n$

Insertion sort typically has a smaller constant factor than merge sort so

$$c_1 < c_2$$

# The Role of Algorithms in Computing

Insertion Sort  $c_1 n^2$

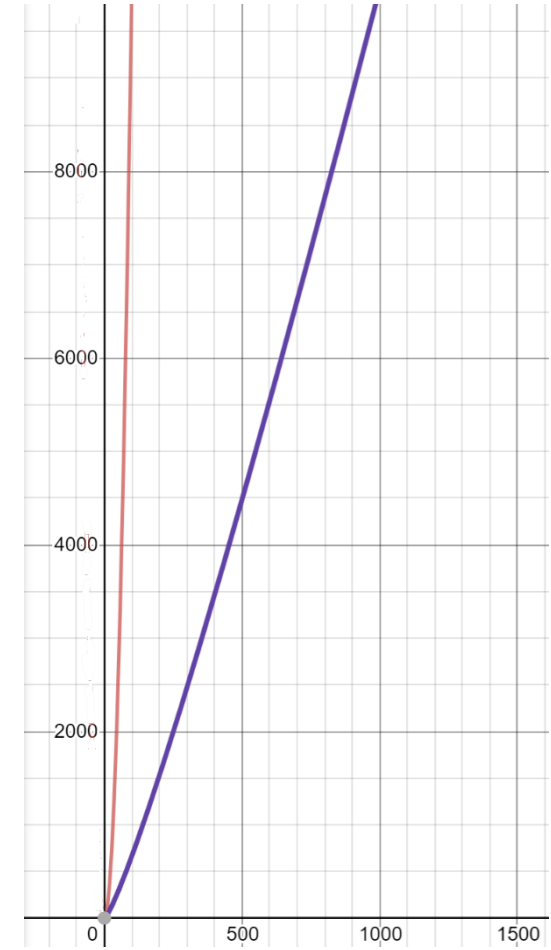
Merge Sort  $c_2 n \log_2 n$

Let's compare  $n^2$  to  $n \log_2 n$

If  $n$  is 10, then  $n^2$  is 100 and  $n \log_2 n$  is  $\sim 33$ .

If  $n$  is 1000, then  $n^2$  is 1,000,000 and  $n \log_2 n$  is  $\sim 9966$

If  $n$  is 1,000,000, then  $n^2$  is 1,000,000,000,000 and  $n \log_2 n$  is  $\sim 19,931,569$





# The Role of Algorithms in Computing

Insertion Sort       $c_1 n^2$

$$c_1 < c_2$$

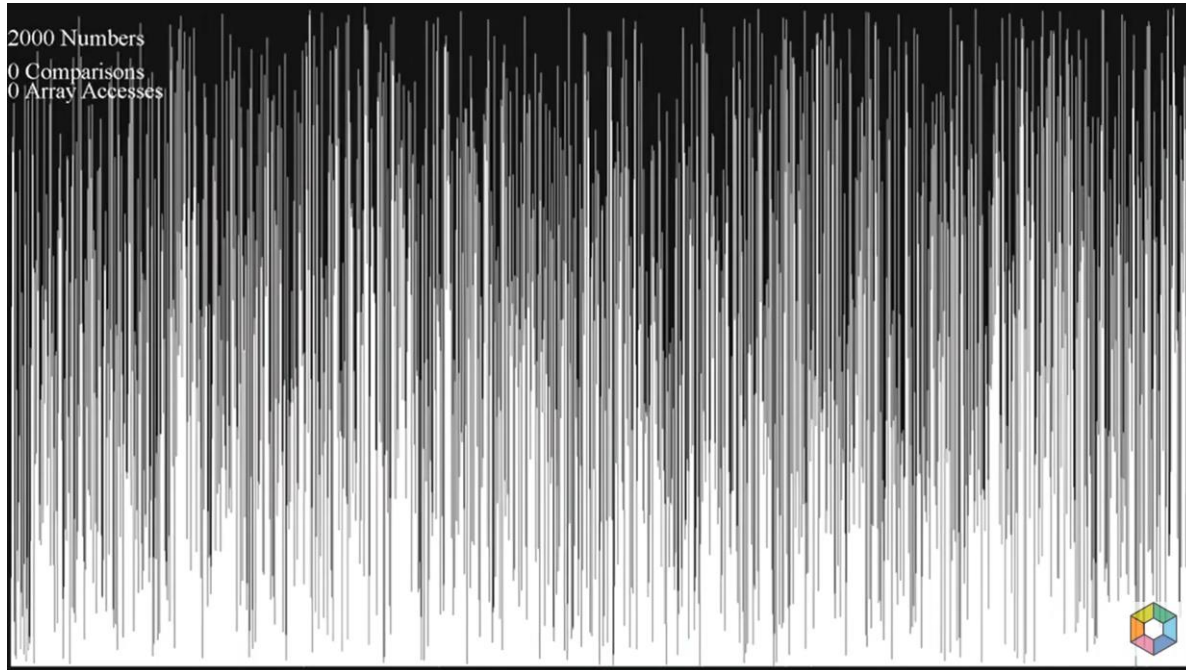
Merge Sort       $c_2 n \log_2 n$

If  $n$  is 1,000,000, then  $n^2$  is 1,000,000,000,000 and  $n \log_2 n$  is  $\sim 200$

How much larger would  $c_2$  need to be than  $c_1$  to get these two values even close to each other?

No matter how much smaller  $c_1$  is than  $c_2$ , there will always be a crossover point beyond which merge sort is faster.

# The Role of Algorithms in Computing



Insertion Sort

$$c_1 n^2$$

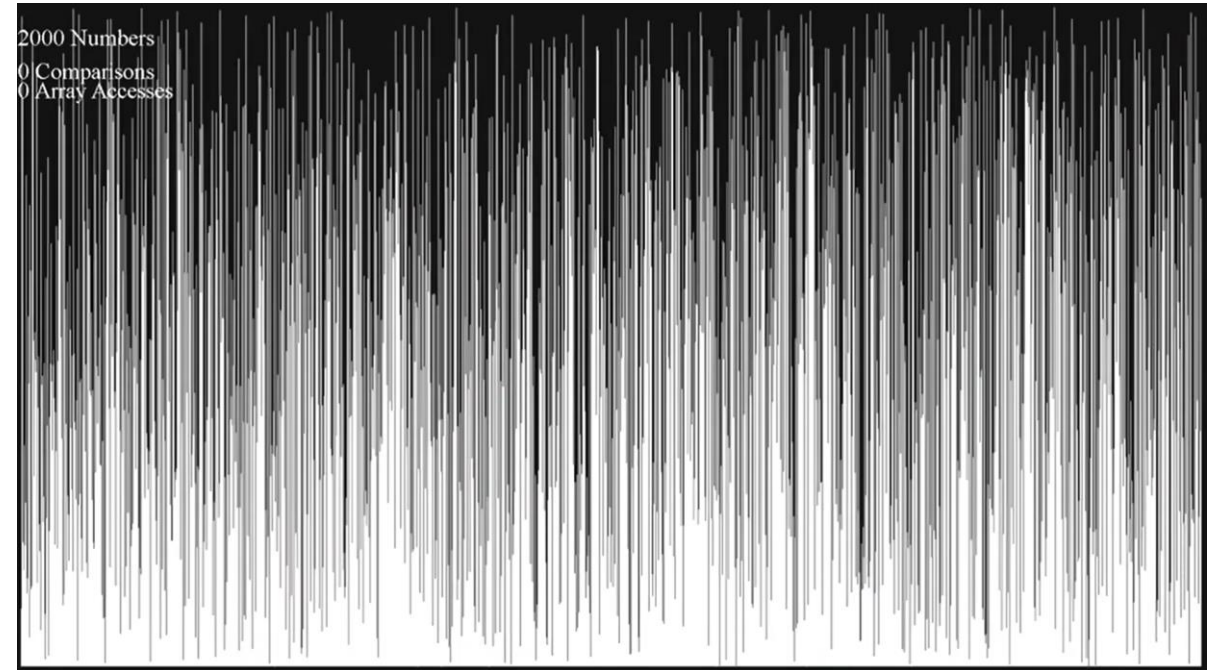
$n = 2000$

$$c_1(2000^2) = c_1(4,000,000)$$

$$4000000c_1 = 2,893,035 \text{ actions}$$

$$c_1 \approx 0.72$$

$$0.72n^2$$



Merge Sort

$$c_2 n \log_2 n$$

$n = 2000$

$$c_2(2000)\log_2(2000) \approx c_2(2000)(10.97) \approx c_2(21,940)$$

$$21940c_2 = 63,327 \text{ actions}$$

$$c_2 \approx 2.89$$

$$2.89n \log_2 n$$

$c_1$  is less than  $c_2$

# The Role of Algorithms in Computing

Suppose we are comparing implementations of insertion sort and merge sort on the same machine. For inputs of size  $n$ , insertion sort runs in  $8n^2$  steps while merge sort runs in  $64n\log_2 n$  steps.

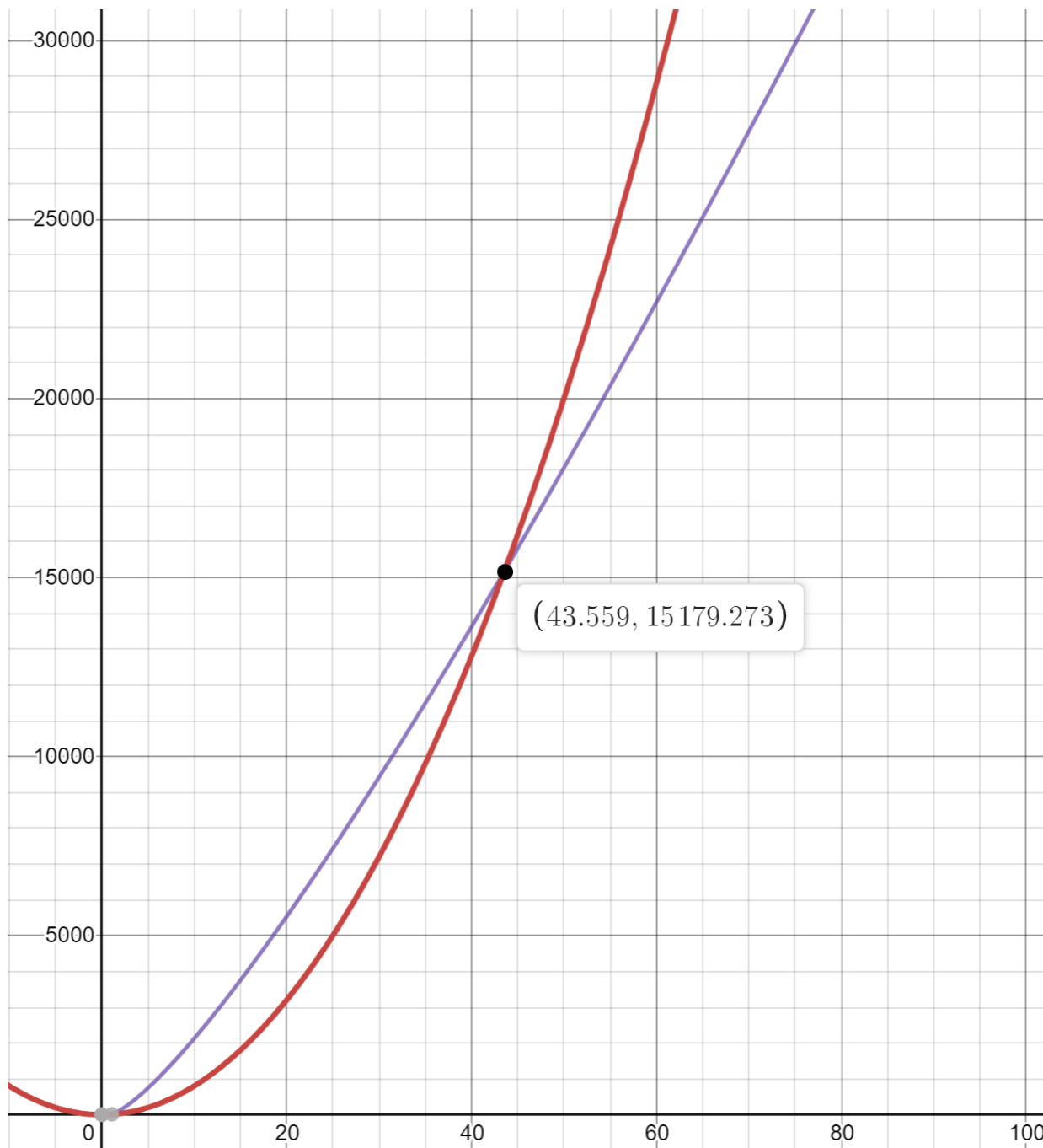
For which values of  $n$  does insertion sort beat merge sort?

Insertion	Merge	Insertion	Merge	Insertion	Merge
$8n^2$	$64n\log_2 n$	$8n^2$	$64n\log_2 n$	$8n^2$	$64n\log_2 n$
$n = 2$	$n = 2$	$n = 32$	$n = 32$	$n = 64$	$n = 64$
$8(2)^2$	$64(2)(\log_2 2)$	$8(32)^2$	$64(32)(\log_2 32)$	$8(64)^2$	$64(64)(\log_2 64)$
32	128	8192	10240	32,768	24,576

# The Role of Algorithms in Computing

So for what input of size  $n$ , does an insertion sort running in  $8n^2$  steps lose to a merge sort that runs in  $64n\log_2 n$  steps?

If we graph the two equations, we can physically see where the run time for insertion sort crosses over the run time for merge sort.



## Insertion

$$8n^2$$

$$n = 43$$

$$8(43)^2$$

$$14,792$$

## Merge

$$64n \log_2 n$$

$$n = 43$$

$$64(43)(\log_2 43)$$

$$\sim 14,933$$

## Insertion

$$8n^2$$

$$n = 44$$

$$8(44)^2$$

$$15,488$$

## Merge

$$64n \log_2 n$$

$$n = 44$$

$$64(44)(\log_2 44)$$

$$\sim 15,374$$

# The Role of Algorithms in Computing

What happens if we remove the coefficients from the equations so that we are comparing an insertion sort running in  $n^2$  steps to a merge sort that runs in  $n\log_2 n$  steps?

For which values of  $n$  does insertion sort beat merge sort?

Insertion	Merge	Insertion	Merge	Insertion	Merge
$n^2$	$n\log_2 n$	$n^2$	$n\log_2 n$	$n^2$	$n\log_2 n$
$n = 2$	$n = 2$	$n = 32$	$n = 32$	$n = 64$	$n = 64$
$(2)^2$	$(2)(\log_2 2)$	$(32)^2$	$(32)(\log_2 32)$	$(64)^2$	$(64)(\log_2 64)$
4	2	1024	160	4096	284

We can see from graphing the two equations, **insertion sort** is never better than **merge sort** when no coefficients are involved.



# Insertion Sort

## Sorting Problem

Input : A sequence of  $n$  numbers  $\langle a_1, a_2, \dots, a_n \rangle$ .

Output : A permutation (reordering)  $\langle a'_1, a'_2, \dots, a'_n \rangle$  of the input sequence such that  $a'_1 \leq a'_2 \leq \dots \leq a'_n$ .

One way to solve this sorting problem is with the insertion sort algorithm.



# Insertion Sort

Insertion Sort algorithm sorts the input numbers in place.

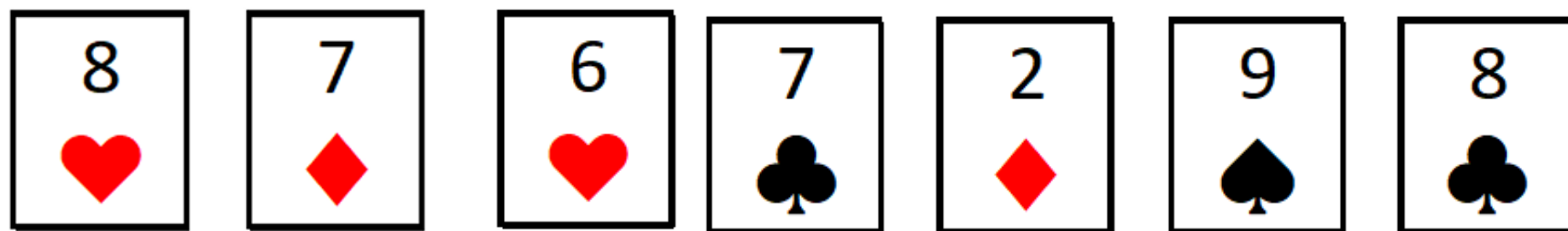
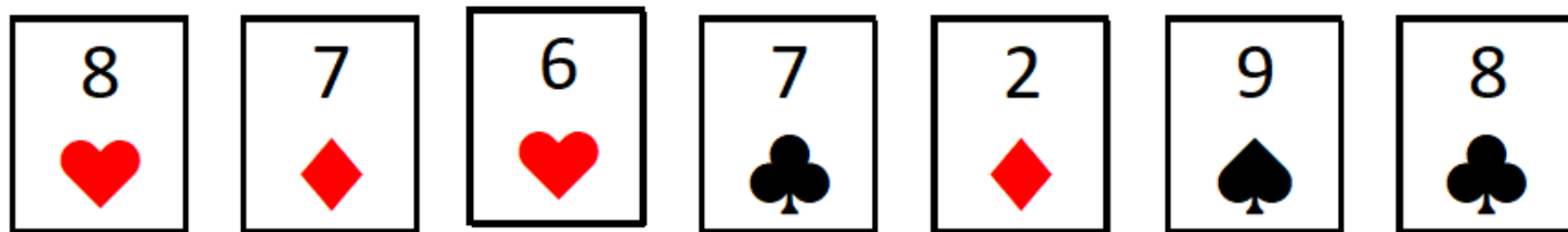
It rearranges the numbers within the array.

The input array of an insertion sort also contains the sorted output when the sort completes.

We will refer to the array element that we are moving around as the "key".

# Insertion Sort

4 3 2 10 12 1 5 6



# Insertion Sort

12, 11, 13, 5, 6

Let us loop for  $i = 1$  (2<sup>nd</sup> element of the array) to 4 (last element of the array)

$i = 1$  Since 11 is smaller than 12, move 12 and insert 11 before 12  
11, 12, 13, 5, 6

$i = 2$  13 will remain at its position as all elements in  $A[0..i-1]$  are smaller than 13  
11, 12, 13, 5, 6

$i = 3$  5 will move to the beginning and all other elements from 11 to 13 will move one position ahead of their current position.  
5, 11, 12, 13, 6

$i = 4$  6 will move to position after 5, and elements from 11 to 13 will move one position ahead of their current position.  
5, 6, 11, 12, 13

# Insertion Sort



# Insertion Sort

$n$  is the number of elements to sort

for  $j = 2$  to  $n$

key =  $A[j]$

// Insert  $A[j]$  into sorted sequence

$i = j - 1$ ;

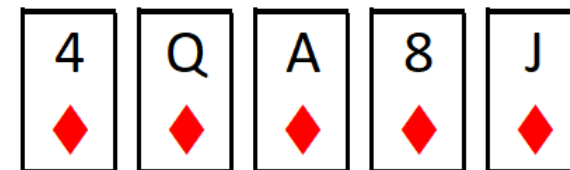
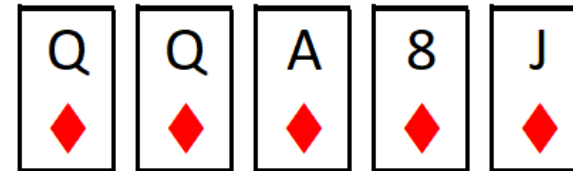
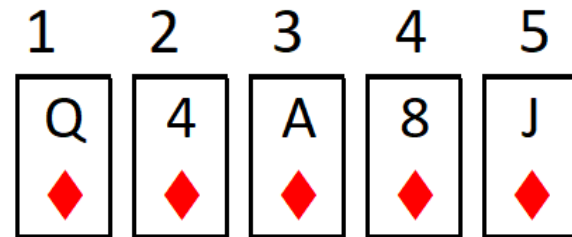
while  $i > 0$  and  $A[i] > \text{key}$

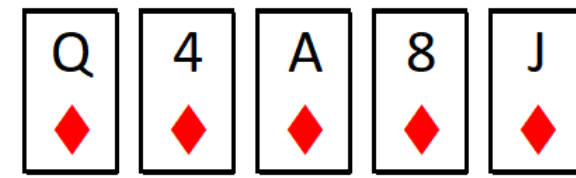
$A[i + 1] = A[i]$ ;

$i = i - 1$ ;

$A[i + 1] = \text{key}$ ;

j	key	i	A[i]	A[i+1]





# Insertion Sort

$n$  is the number of elements to sort

for  $j = 2$  to  $n$

key =  $A[j]$

// Insert  $A[j]$  into sorted sequence

$i = j - 1$ ;

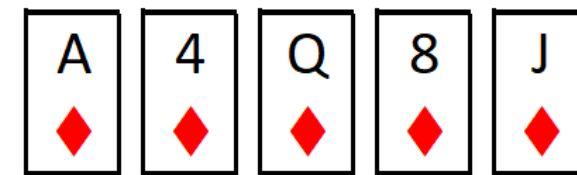
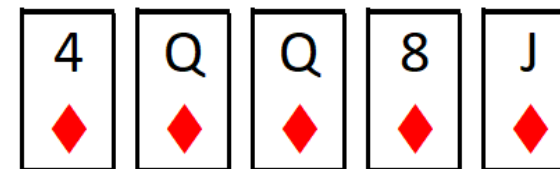
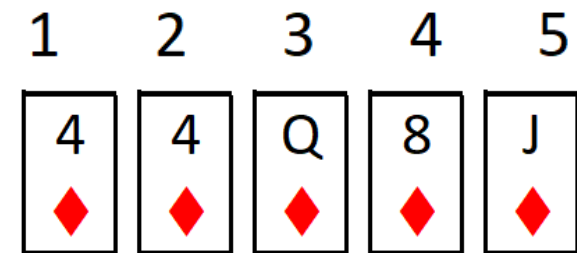
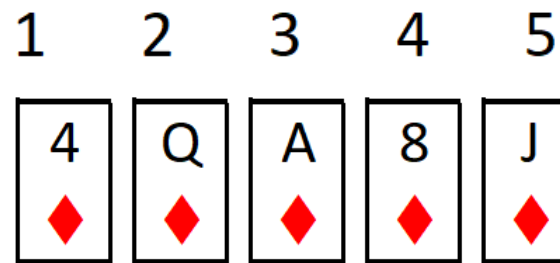
while  $i > 0$  and  $A[i] > \text{key}$

$A[i + 1] = A[i]$ ;

$i = i - 1$ ;

$A[i + 1] = \text{key}$ ;

j	key	i	A[i]	A[i+1]



$n$  is the number of elements to sort

for  $j = 2$  to  $n$

key =  $A[j]$

// Insert  $A[j]$  into sorted sequence

$i = j - 1$ ;

while  $i > 0$  and  $A[i] > \text{key}$

$A[i + 1] = A[i]$ ;

$i = i - 1$ ;

$A[i + 1] = \text{key}$ ;

# Insertion Sort

j	key	i	A[i]	A[i+1]



# Analysis of Insertion Sort

How long an insertion sort runs depends on several factors...

- characteristics of the input

  - amount of it

  - how sorted it already is

In general, the time taken by an algorithm grows with the size of the input.

The "running time" of a program is described as a function of the "input size".

# Analysis of Insertion Sort

How "input size" is defined depends on the problem...

For sorting, the number of items in the input (array size)

For multiplying two integers, the input size is the total number of bits needed to represent the input

For a graph problem, the input size can be described by the numbers of vertices and edges in the graph.

# Analysis of Insertion Sort

The "running time" of an algorithm on a particular input is the number of primitive operations or "steps" executed.

These steps should be as machine-independent as possible.

Let's assume that a constant amount of time is required to execute each line of our pseudocode.

One line may take a different amount of time than another, but each execution of line  $i$  takes the same amount of time  $c_i$ .

The for loop will start execution with a value of 2 and will continue to loop until  $n$  where  $n$  is the number of items to be sorted. This value is  $j$  in our pseudocode.

Let  $t_j$  be the number of times that the while loop test is executed for that value of  $j$ .

Remember that when a for or while loop exits in the usual way—due to the test in the loop header—the test is executed one time more than the loop body.

# Analysis of Insertion Sort

	cost	times	
1 for j = 2 to n	$c_1$	$n$	
2 key = A[j]	$c_2$	$n - 1$	Execute one less time than loop
3 // Insert A[j] into sorted part			
4 i = j - 1	$c_4$	$n - 1$	
5 while i > 0 and A[i] > key	$c_5$	$\sum_{j=2}^n t_j$	Sum over the loop
6 A[i + 1] = A[i];	$c_6$	$\sum_{j=2}^n (t_j - 1)$	
7 i = i - 1	$c_7$	$\sum_{j=2}^n (t_j - 1)$	Execute one less time than loop
8 A[i + 1] = key	$c_8$	$n - 1$	

# Analysis of Insertion Sort

To compute  $T(n)$ , the running time of Insertion Sort on an input of  $n$  values, we sum the products of the cost and times columns

$$T(n) = c_1n + c_2(n - 1) + c_4(n - 1) +$$

$$c_5\sum_{j=2}^n t_j +$$

$$c_6\sum_{j=2}^n (tj - 1) +$$

$$c_7\sum_{j=2}^n (tj - 1) +$$

$$c_8(n - 1)$$

# Analysis of Insertion Sort

## Best Case Scenario

The array is already sorted which means that we always find that

$$A[j] \leq \text{key}$$

every time the while loop is tested for the first time (when  $i = j - 1$ ).

$t_j$  is the number of times that the while loop test is executed for that value of  $j$ .

In the best case scenario,  $t_j$  is 1 because the while loop tests executes only once for each pass through the for loop (each value of  $j$ ).

## Best Case      5 6 11 12 13

Before  $j=2$  loop      5 6 11 12 13

$c_6$  and  $c_7$  did not execute

After  $j=2$  loop      5 6 11 12 13

Before  $j=3$  loop      5 6 11 12 13

$c_6$  and  $c_7$  did not execute

After  $j=3$  loop      5 6 11 12 13

Before  $j=4$  loop      5 6 11 12 13

$c_6$  and  $c_7$  did not execute

After  $j=4$  loop      5 6 11 12 13

Before  $j=5$  loop      5 6 11 12 13

$c_6$  and  $c_7$  did not execute

After  $j=5$  loop      5 6 11 12 13

Final value of  $j$  is 6

$c_1$  executed 5 times –  $c_1$  executed  $n$  times ( $n$  is the number of array elements)

$c_2, c_4, c_5, c_8$  executed 4 times -  $c_2, c_4, c_5, c_8$  executed  $n-1$  times

$c_6$  and  $c_7$  did not execute

$c_1$     for  $j = 2$  to  $n$

$c_2$        $\text{key} = A[j]$

// Insert  $A[j]$  into sorted part

$c_4$        $i = j - 1$

$c_5$       while  $i > 0$  and  $A[i] > \text{key}$

$c_6$            $A[i + 1] = A[i];$

$c_7$            $i = i - 1$

$c_8$        $A[i + 1] = \text{key}$

# Analysis of Insertion Sort

## Best Case Scenario

Running time

$$\begin{aligned} T(n) = & c_1n + c_2(n-1) + c_4(n-1) + \\ & c_5\sum_{j=2}^n t_j + \\ & c_6\sum_{j=2}^n (tj - 1) + \\ & c_7\sum_{j=2}^n (tj - 1) + \\ & c_8(n-1) \end{aligned}$$

can be simplified to

$$T(n) = c_1n + c_2(n-1) + c_4(n-1) + c_5(n-1) + c_8(n-1)$$



# Analysis of Insertion Sort

## Best Case Scenario

Running time

$$T(n) = c_1n + c_2(n-1) + c_4(n-1) + c_5(n-1) + c_8(n-1)$$

can be further simplified to

$$\begin{aligned} T(n) &= c_1n + c_2n - c_2 + c_4n - c_4 + c_5n - c_5 + c_8n - c_8 \\ &= c_1n + c_2n + c_4n + c_5n + c_8n - c_2 - c_4 - c_5 - c_8 \\ &= (c_1 + c_2 + c_4 + c_5 + c_8)n - (c_2 + c_4 + c_5 + c_8) \end{aligned}$$

$$T(n) = an + b$$

where constants  $a$  and  $b$  depend on the statement costs  $c_i$

$T(n)$  is a linear function

# Analysis of Insertion Sort

## Worst Case Scenario

The array is in reverse sorted order.

Always find that

$$A[j] > key$$

in while loop test.

Have to compare *key* with all elements to the left of the *j*th position which means always comparing with *j* - 1 elements

Before j=2 loop        13 12 11 6 5  
i before 1 and i after 0

After j=2 loop        12 13 11 6 5

Before j=3 loop        12 13 11 6 5  
i before 2 and i after 1  
i before 1 and i after 0

After j=3 loop        11 12 13 6 5

Before j=4 loop        11 12 13 6 5  
i before 3 and i after 2  
i before 2 and i after 1  
i before 1 and i after 0

After j=4 loop        6 11 12 13 5

Before j=5 loop        6 11 12 13 5  
i before 4 and i after 3  
i before 3 and i after 2  
i before 2 and i after 1  
i before 1 and i after 0

After j=5 loop        5 6 11 12 13

Final value of j is 6

$c_1, c_2, c_4, c_8$

$c_5, c_6, c_7, c_5$

$c_1, c_2, c_4, c_8$

$c_5, c_6, c_7$

$c_5, c_6, c_7, c_5$

$c_1, c_2, c_4, c_8$

$c_5, c_6, c_7$

$c_5, c_6, c_7$

$c_5, c_6, c_7, c_5$

$c_1, c_2, c_4, c_8$

$c_5, c_6, c_7$

$c_5, c_6, c_7$

$c_5, c_6, c_7$

$c_5, c_6, c_7, c_5$

$c_1$

**Worst Case    13,12,11,6,5**

$c_1$  for j = 2 to n

$c_2$         key = A[j]

// Insert A[j] into sorted part

$c_4$         i = j - 1

$c_5$         while i > 0 and A[i] > key

$c_6$         A[i + 1] = A[i];

$c_7$         i = i - 1

$c_8$         A[i + 1] = key

Before j=2 loop      13 12 11 6 5  
i before 1 and i after 0

After j=2 loop      12 13 11 6 5

Before j=3 loop      12 13 11 6 5  
i before 2 and i after 1  
i before 1 and i after 0

After j=3 loop      11 12 13 6 5

Before j=4 loop      11 12 13 6 5  
i before 3 and i after 2  
i before 2 and i after 1  
i before 1 and i after 0

After j=4 loop      6 11 12 13 5

Before j=5 loop      6 11 12 13 5  
i before 4 and i after 3  
i before 3 and i after 2  
i before 2 and i after 1  
i before 1 and i after 0

After j=5 loop      5 6 11 12 13

Final value of j is 6

$C_1, C_2, C_4, C_8$

$C_5, C_6, C_7, C_5$

$C_1, C_2, C_4, C_8$

$C_5, C_6, C_7$

$C_5, C_6, C_7, C_5$

$C_1, C_2, C_4, C_8$

$C_5, C_6, C_7$

$C_5, C_6, C_7$

$C_5, C_6, C_7, C_5$

$C_1, C_2, C_4, C_8$

$C_5, C_6, C_7$

$C_5, C_6, C_7$

$C_5, C_6, C_7$

$C_5, C_6, C_7, C_5$

$C_1$

**Worst Case      13,12,11,6,5**

$C_1$

5 times    n times

$C_2, C_4, C_8$

4 times    n – 1 times

$C_5$

14 times

1 + 1 more to fail

2 + 1 more to fail

3 + 1 more to fail

4 + 1 more to fail

$C_6, C_7$

1x per while loop

2x per while loop

3x per while loop

4x per while loop

# Analysis of Insertion Sort

$t_j$  is the number of times the while loop runs

## Worst Case Scenario

$c_1$

5 times     $n$  times

Running time

$c_2, c_4, c_8$

4 times     $n - 1$  times

$$T(n) = c_1n + c_2(n - 1) + c_4(n - 1) + c_5 \sum_{j=2}^n t_j + c_6 \sum_{j=2}^n (t_j - 1) + c_7 \sum_{j=2}^n (t_j - 1) + c_8(n - 1)$$

$c_5$

14 times

1 + 1 more to fail

2 + 1 more to fail

3 + 1 more to fail

4 + 1 more to fail

$c_6, c_7$

1x per while loop

2x per while loop

3x per while loop

4x per while loop

The formula for  $c_5$  needs a slight change to accommodate the "1 more to fail" pass

# Analysis of Insertion Sort

## Worst Case Scenario

$t_j$  is the number of times the while loop runs

Since the while loop exits because  $i$  reaches 0, there is one additional test for each pass which equals  $j$ ; therefore,

$$t_j = j$$

So  $c_5 \sum_{j=2}^n t_j$  is now  $c_5 \sum_{j=2}^n j$

$c_1$

5 times     $n$  times

$c_2, c_4, c_8$

4 times     $n - 1$  times

$c_5$

14 times

1 + 1 more to fail

2 + 1 more to fail

3 + 1 more to fail

4 + 1 more to fail

$c_6, c_7$

1x per while loop

2x per while loop

3x per while loop

4x per while loop

# Analysis of Insertion Sort

## Worst Case Scenario

$t_j$  is the number of times the while loop runs

So  $c_5 \sum_{j=2}^n t_j$  is now  $c_5 \sum_{j=2}^n j$  which is  $c_5 (\sum_{j=1}^n j - 1)$

$\sum_{j=1}^n j$  is an arithmetic series that equals  $\frac{n(n+1)}{2}$

$c_5 \sum_{j=2}^n t_j$  equals  $c_5 (\sum_{j=1}^n j - 1)$  and  $\sum_{j=1}^n j - 1$  equals  $\frac{n(n+1)}{2} - 1$

So  $c_5 \sum_{j=2}^n j$  can be expressed as  $c_5 (\frac{n(n+1)}{2} - 1)$

$$\frac{n(n+1)}{2} - 1 \text{ when } n = 5$$

$$\frac{5(5+1)}{2} - 1 = 14$$

$c_5$

14 times

2+3+4+5

# Analysis of Insertion Sort

## Worst Case Scenario

$t_j$  is the number of times the while loop runs

$$c_6 \sum_{j=2}^n (t_j - 1) \text{ and } c_7 \sum_{j=2}^n (t_j - 1)$$

$$\sum_{j=2}^n t_j = \sum_{j=2}^n j \text{ and } \sum_{j=2}^n (t_j - 1) = \sum_{j=2}^n (j - 1)$$

$$\sum_{k=1}^a k \text{ is an arithmetic series that equals } \frac{a(a+1)}{2}$$

$$\text{If we let } k = j - 1, \text{ then } \sum_{j=2}^n (j - 1) \text{ equals } \sum_{k=1}^{n-1} k \text{ which equals } \frac{n(n-1)}{2}$$

$$\text{So } \sum_{j=2}^n (t_j - 1) \text{ can be expressed as } \frac{n(n-1)}{2}$$

$$\frac{n(n-1)}{2} \text{ when } n = 5$$

$$\frac{5(5-1)}{2} = 10$$

$c_6, c_7$

1x per while loop

2x per while loop

3x per while loop

4x per while loop



# Analysis of Insertion Sort

$t_j$  is the number of  
times the while  
loop runs

## Worst Case Scenario

Running time

$$\begin{aligned} T(n) = & c_1n + c_2(n-1) + c_4(n-1) + c_5 \sum_{j=1}^n j - 1 + \\ & c_6 \sum_{j=2}^n (t_j - 1) + \\ & c_7 \sum_{j=2}^n (t_j - 1) + \\ & c_8(n-1) \end{aligned} \quad \begin{aligned} T(n) = & c_1n + c_2(n-1) + c_4(n-1) + \\ & c_5 \left( \frac{n(n+1)}{2} - 1 \right) + \\ & c_6 \left( \frac{n(n-1)}{2} \right) + \\ & c_7 \left( \frac{n(n-1)}{2} \right) + \\ & c_8(n-1) \end{aligned}$$

# Analysis of Insertion Sort

$t_j$  is the number of  
times the while  
loop runs

## Worst Case Scenario

Running time

$$\begin{aligned} T(n) = & c_1n + c_2(n-1) + c_4(n-1) + \\ & c_5\left(\frac{n(n+1)}{2} - 1\right) + \\ & c_6\left(\frac{n(n-1)}{2}\right) + \\ & c_7\left(\frac{n(n-1)}{2}\right) + \\ & c_8(n-1) \end{aligned}$$

$$\begin{aligned} T(n) = & c_1n + c_2n - c_2 + c_4n - c_4 + \\ & c_5\frac{n^2}{2} + c_5\frac{n}{2} - c_5 \\ & c_6\frac{n^2}{2} + c_6\frac{n}{2} \\ & c_7\frac{n^2}{2} + c_7\frac{n}{2} + \\ & c_8n - c_8 \end{aligned}$$

# Analysis of Insertion Sort

$t_j$  is the number of times the while loop runs

## Worst Case Scenario

Running time

$$T(n) = c_1n + c_2n - c_2 + c_4n - c_4 + c_5\frac{n^2}{2} + c_5\frac{n}{2} - c_5 c_6\frac{n^2}{2} + c_6\frac{n}{2} c_7\frac{n^2}{2} + c_7\frac{n}{2} + c_8n - c_8$$

$$T(n) = \left(\frac{c_5}{2} + \frac{c_6}{2} + \frac{c_7}{2}\right)n^2 + \left(c_1 + c_2 + c_4 + \frac{c_5}{2} + \frac{c_6}{2} + \frac{c_7}{2} + c_8\right)n - (c_2 + c_4 + c_5 + c_8)$$

$$T(n) = an^2 + bn + c$$

where constants  $a$ ,  $b$  and  $c$  depend on the statement costs  $c_i$

$T(n)$  is a quadratic function

# Worst-case and Average-case Analysis

So we found the best case (array was already sorted) running time and the worst case (array was reverse sorted) running time.

Best case running time was linear.

Worst case running time was quadratic.

What about the average case?

# Worst-case and Average-case Analysis

What is an average case?

The ability to create an average case is limited because it may not be apparent what constitutes an "average" input for a particular problem.

Suppose that we randomly choose  $n$  numbers as the input to insertion sort in order create an "average" input.

On average, the key in  $A[j]$  is less than half the elements in  $A[1..j-1]$  and it's greater than the other half.

On average, the while loop has to look halfway through the sorted subarray  $A[1..j-1]$  to decide where to drop *key*.

$$t_j \approx j/2$$

Although the average-case running time is approximately half of the worst-case running time, it's still a quadratic function of  $n$ .

# Worst-case and Average-case Analysis

We usually concentrate on finding the worst-case running time: the longest running time for any input of size  $n$ .

## Reasons

The worst-case running time gives a guaranteed upper bound on the running time for any input.

For some algorithms, the worst case occurs often.

For example, when searching, the worst case often occurs when the item being searched for is not present, and searches for absent items may be frequent.

Why not analyze the average case? Because it's often about as bad as the worst case. Although the average-case running time is approximately half of the worst-case running time, it's still a quadratic function of  $n$

# Order of Growth

We used some simplifying abstractions to ease our analysis of Insertion Sort.

- We ignored the actual cost of each statement by using  $c_i$
- We went a step further by using constants  $a$ ,  $b$  and  $c$  to represent different groupings of  $c_i$

We are going to use one more abstraction to simplify our analysis

Since it is the rate of growth or order of growth of the running time that really interests us, we will only consider the leading term of the formula – we ignore the lower-order terms and the leading term's constant coefficient.

# Order of Growth

For example, our worst case running time of insertion sort is

$$an^2 + bn + c$$

Ignoring the lower-order terms and the leading term's constant coefficient gives us

$$n^2$$

But we cannot say that the worst-case running time  $T(n)$  equals  $n^2$

It grows like  $n^2$  but it does not **equal**  $n^2$ .



# Order of Growth

$$an^2 + bn + c$$

When  $n = 1$ , the lower-order terms and the leading term's constant coefficient have more weight/influence.

The value of  $c$  for example, could easily overshadow  $n$  when  $n$  is small.

But, when  $n$  gets larger like  $n = 1000000$

$n^2$  is 1000000000000 and the values of  $a$ ,  $b$ ,  $c$  and even the lower order  $n$  will have much less impact – a small enough effect to ignore.

# Order of Growth

It grows like  $n^2$  but it does not **equal**  $n^2$ .

To show growth without equivalency, the following notation is used

Insertion sort has a worst-case running time of  $\Theta(n^2)$

This is pronounced as "theta of  $n$ -squared"

We usually consider one algorithm to be more efficient than another if its worst case running time has a smaller order of growth.

# Exercise

Express the function

$$\frac{n^3}{1000} - 100n^2 - 100n + 3$$

in terms of  $\Theta$ -notation.

$$\Theta(n^3)$$

$n$	$\frac{n^3}{1000} - 100n^2 - 100n + 3$	$n^3$
1	-197	1
10	-10,996	1,000
100	-1,008,997	1,000,000
1,000	-99,099,997	1,000,000,000
10,000	-9,000,999,997	1,000,000,000,000
100,000	-9,999,997	1,000,000,000,000,000
1,000,000	899,999,900,000,003	1,000,000,000,000,000,000
10,000,000	989,999,999,000,000,000	1,000,000,000,000,000,000,000
100,000,000	998,999,999,990,000,000,000	1,000,000,000,000,000,000,000,000
1,000,000,000	999,899,999,999,900,000,000,000	1,000,000,000,000,000,000,000,000,000
10,000,000,000	999,989,999,999,999,000,000,000,000	1,000,000,000,000,000,000,000,000,000,000
100,000,000,000	999,999,000,000,000,000,000,000,000,000	1,000,000,000,000,000,000,000,000,000,000,000

# Asymptotic Notation

We use big- $\Theta$  notation to asymptotically bound the growth of a running time to within constant factors above and below.

Sometimes we want to bound from only above.

For example, although the worst-case running time of binary search is

$\Theta(\log_2 n)$

it would be incorrect to say that binary search runs in  $\Theta(\log_2 n)$  time in *all* cases.

# Asymptotic Notation

What if we find the target value upon the first guess?

Then it runs in  $\Theta(1)$  time.

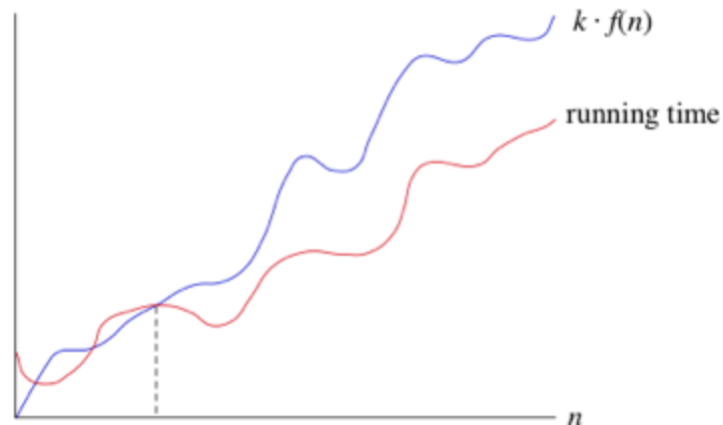
The running time of binary search is never worse than  $\Theta(\log_2 n)$ , but it's sometimes better.

A form of asymptotic notation called "big-O" notation means "the running time grows at most this much, but it could grow more slowly."

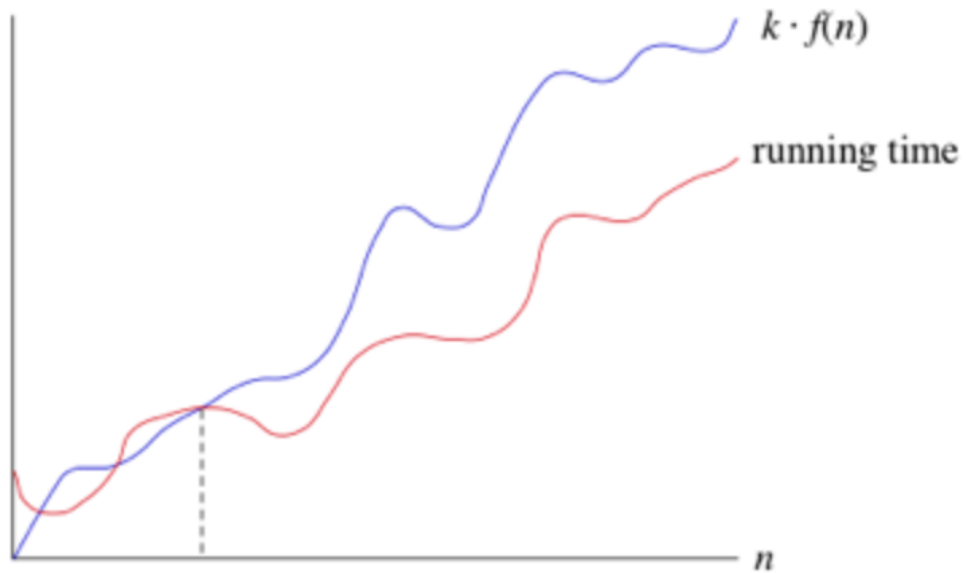
# Asymptotic Notation

"big-O" notation means "the running time grows at most this much, but it could grow more slowly."

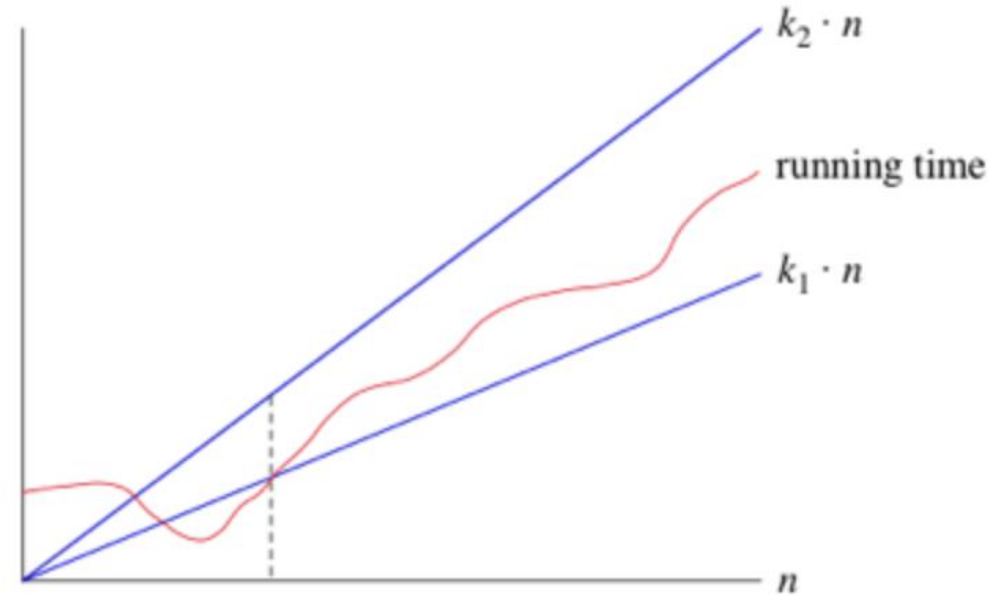
We use big-O notation for **asymptotic upper bounds**, since it bounds the growth of the running time from above for large enough input sizes.



# Asymptotic Notation



Big O  
asymptotic upper bound



Big  $\Theta$   
asymptotically tight bound

# Asymptotic Notation

Because big-O notation gives only an asymptotic upper bound, and not an asymptotically tight bound, we can make statements that at first glance seem incorrect but are technically correct.

We have said that the worst-case running time of binary search is  $\Theta(\log_2 n)$  time.

It is still correct to say that binary search runs in  $O(n)$ .

That's because the running time grows no faster than a constant times  $n$ . In fact, it grows slower ( $\log_2 n$ )



# Asymptotic Notation

If you have \$10 in your pocket, you can honestly say "I have an amount of money in my pocket, and I guarantee that it's no more than one million dollars."

Your statement is absolutely true but not terribly precise.

One million dollars is an upper bound on \$10, just as  $O(n)$  is an upper bound on the running time of binary search.  $O(n)$  is not very precise but it is still correct.

# Asymptotic Notation

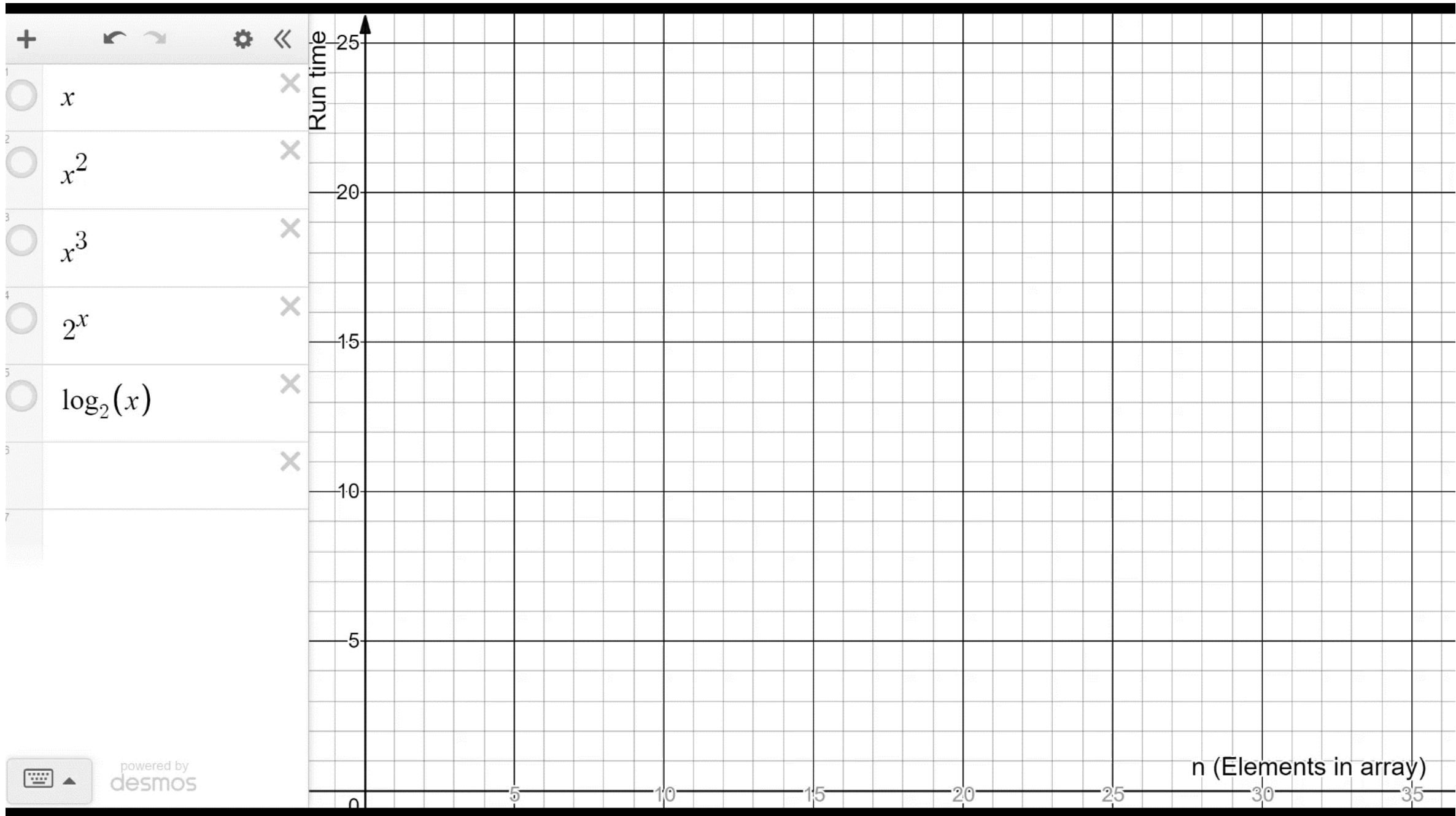
The worst-case running time of binary search is  $\Theta(\log_2 n)$  time but it would also be accurate (not precise) to state that binary search has a run time of  $O(n)$ ,  $O(n^2)$ ,  $O(n^3)$ , and  $O(2^n)$ .

For example, if  $n = 4$ ...

$$\log_2 n < O(n) < O(n^2) < O(2^n) < O(n^3) \qquad 2 \leq 4 \leq 16 \leq 16 \leq 64$$

We don't have much use for these correct but imprecise big O representations.

By precise, we mean the notation that gives us the best idea of the actual run time.



# Asymptotic Notation

If we have a set like this

$\{2, 3, 5, 7, 9, 12, 17, 42\}$

then the tight lower bound is the greatest of all lower bounds

0 is a lower bound

1.99 is a lower bound

-32,567 is a lower bound

2 is a lower bound

Which one of these is the greatest of all lower bounds?

2

# Asymptotic Notation

If we have a set like this

$\{2, 3, 5, 7, 9, 12, 17, 42\}$

then the tight upper bound is the least of all upper bounds

42.001 is an upper bound

4561.99 is an upper bound

932,567 is an upper bound

42 is an upper bound

Which one of these is the least of all upper bounds?

42

# Coding Assignment 1

## Coding Assignment 1 goals

- use conditional compile statements
- use command line parameters
- file handling
- linked list handling
- use the `clock()` function to time functions in your program
- use a provided executable to create large files

# Coding Assignment 1

These are all skills/code you will need for later Coding Assignments. You will be reusing parts of this assignment in other assignments.

Create a program than can open a file listed on the command line, read through that file, write each line from the file into a linked list, print the linked list and free the linked list. You will time each step and count and sum during each step.

# Coding Assignment 1

## Linked Lists

I have taken my 3 lecture recordings on Linked Lists from my CSE 1320 class and combined them into 1 video. It is the recorded Teams lecture that goes with the Linked List slides.

The slides and the video are posted under Review Materials.



# Coding Assignment 1

## Creating your TestFile.txt

Download the file, FileGenerator.e, from Canvas to where you are compiling code on your PC.

The executable should run on your PC with no issues. Do NOT compile it – just run it.

Run it and it will give you a message about what parameters it expects. Use the file generator to create a file of 10 records. Use this file for testing and submit in your zip as “TestFile.txt”.

# Coding Assignment 1

**./FileGenerator.e**

Run as

```
FileGenerator.e FILENAME.txt xxxx
```

where FILENAME.txt is the name of the file to be created and  
xxxx is the total number of random numbers to generate

**./FileGenerator.e TestFile.txt 10**

Enter your student id (all 10 digits) : 1000074079

# Using Recursion

The programs we've discussed are generally structured as functions that call one another in a disciplined, hierarchical manner.

For some types of problems, it's useful to have functions call themselves.

A recursive function is a function that calls itself either directly or indirectly through another function.

Recursion is a complex topic discussed at length in upper-level computer science courses.

# Using Recursion

Recursion occurs when a function or subprogram calls itself or calls a function which in turn calls the original function.

A simple example of a mathematical recursion is factorial

$$1! = 1$$

$$2! = 2 * 1 = 2$$

$$3! = 3 * 2 * 1 = 6$$

$$4! = 4 * 3 * 2 * 1 = 24$$

$$5! = 5 * 4 * 3 * 2 * 1 = 120$$

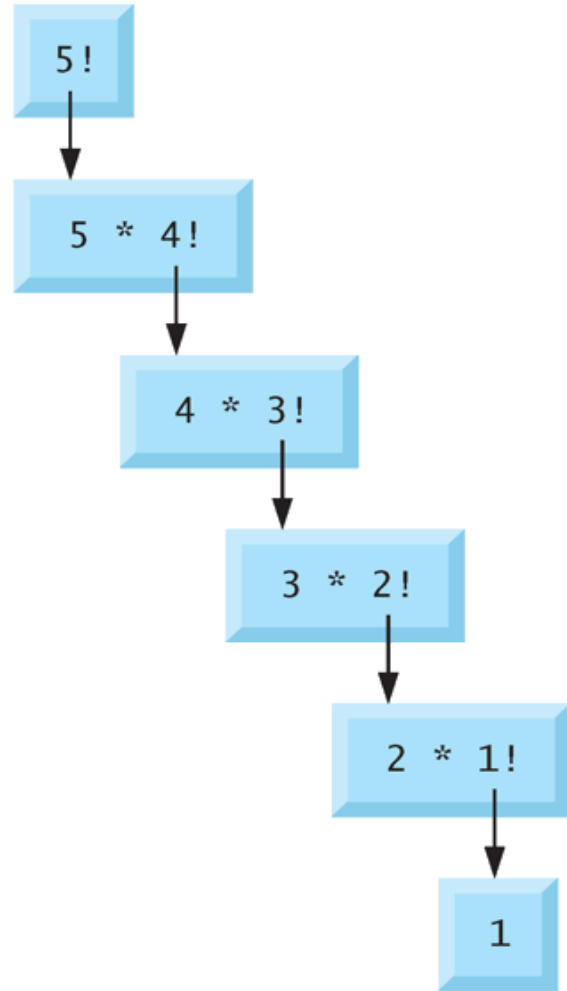
$$n! = n * (n - 1)!$$

# Using Recursion

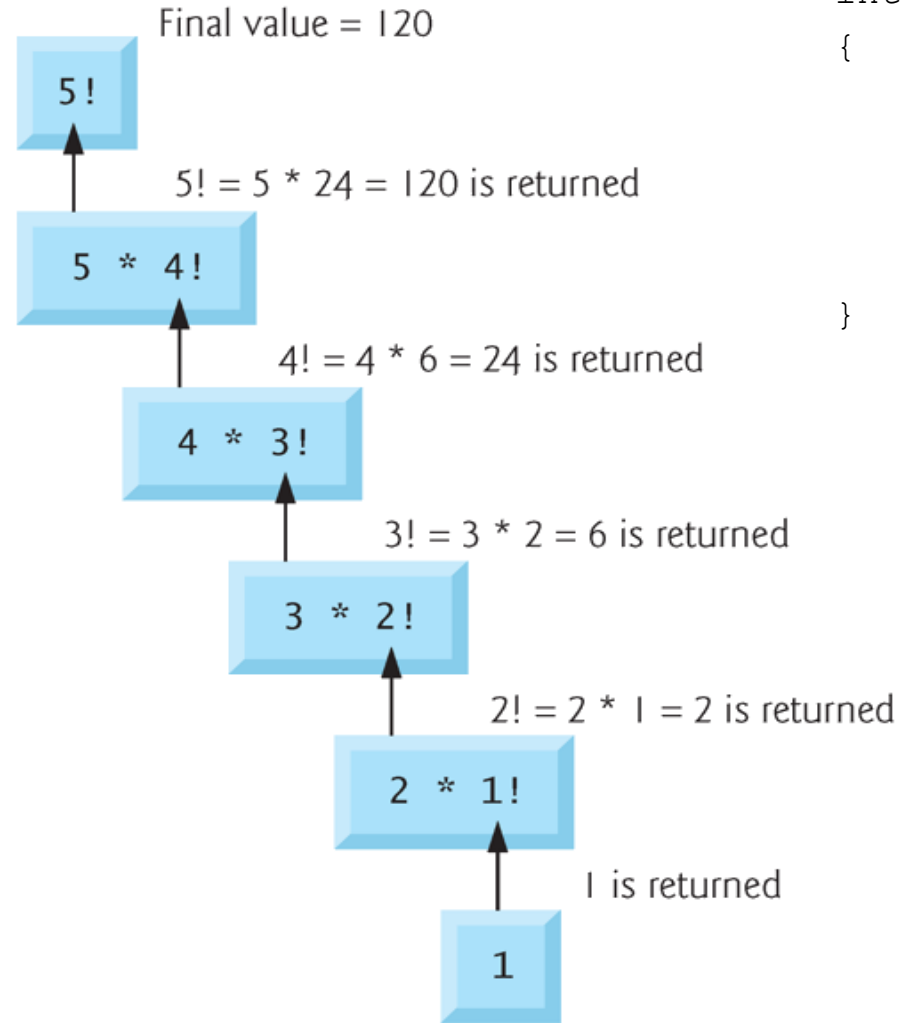
$$n! = n * (n - 1)!$$

```
int factorial(int n)
{
    if (n == 0)
        return 1;
    else
        return (n * factorial(n - 1));
}
```

a) Sequence of recursive calls



b) Values returned from each recursive call



$$n! = n * (n - 1)!$$

```
int factorial(int n)
{
    if (n == 0)
        return 1;
    else
        return (n * factorial(n - 1));
}
```

Recursive evaluation of 5!

```
int main(void)
{
    int input, output;

    printf("Enter an input for the factorial ");
    scanf("%d", &input);

    output = factorial(input);

    printf("The result of %d! is %d\n\n", input, output);

    return 0;
}

int factorial(int n)
{
    if (n == 0)
        return 1;
    else
        return (n * factorial(n - 1));
}
```

Enter an input for the factorial 4 The result of 4! is 24
--

Enter 4

Calls factorial with 4

```
int factorial(int n)
{
    if (n == 0)
        return 1;
    else
        return (n * factorial(n - 1));
}
```

factorial(4)

if 0, then return 1 else return (4 \* factorial(4-1))  
**return (4 \* 6)**

factorial(3)

if 0, then return 1 else return (3 \* factorial(3-1))  
**return (3 \* 2)**

factorial(2)

if 0, then return 1 else return (2 \* factorial(2-1))  
**return (2 \* 1)**

factorial(1)

if 0, then return 1 else return (1 \* factorial(1-1))  
**return (1 \* 1)**

factorial(0)

if 0, then return 1 else return (0 \* factorial(0-1))  
**return 1**

$$4! = 4 * 3 * 2 * 1 = 24$$



# Using Recursion

A function's execution environment includes local variables and parameters and other information like a pointer to the memory containing the global variables.

This execution environment is created every time a function is called.

Recursive functions can use a lot of memory quickly since a new execution environment is created each time the recursive function is called.

(gdb) bt

```
#0  factorial (n=0) at frDemo.c:6
#1  0x000000000004004fd in factorial (n=1) at frDemo.c:9
#2  0x000000000004004fd in factorial (n=2) at frDemo.c:9
#3  0x000000000004004fd in factorial (n=3) at frDemo.c:9
#4  0x000000000004004fd in factorial (n=4) at frDemo.c:9
#5  0x0000000000040053d in main () at frDemo.c:19
```

## After processing n=0

```
#0  0x000000000004004fd in factorial (n=1) at frDemo.c:9
#1  0x000000000004004fd in factorial (n=2) at frDemo.c:9
#2  0x000000000004004fd in factorial (n=3) at frDemo.c:9
#3  0x000000000004004fd in factorial (n=4) at frDemo.c:9
#4  0x0000000000040053d in main () at frDemo.c:19
```

## After processing $n=1$

```
#0  0x0000000000004004fd in factorial (n=2) at frDemo.c:9
#1  0x0000000000004004fd in factorial (n=3) at frDemo.c:9
#2  0x0000000000004004fd in factorial (n=4) at frDemo.c:9
#3  0x00000000000040053d in main () at frDemo.c:19
```

## After processing $n=2$

```
#0  0x0000000000004004fd in factorial (n=3) at frDemo.c:9
#1  0x0000000000004004fd in factorial (n=4) at frDemo.c:9
#2  0x00000000000040053d in main () at frDemo.c:19
```

After processing  $n=3$

```
#0  0x000000000004004fd in factorial (n=4) at frDemo.c:9  
#1  0x0000000000040053d in main () at frDemo.c:19
```

After processing  $n=4$

```
#0  0x0000000000040053d in main () at frDemo.c:19
```

## Recursive Program to Sum Range of Natural Numbers

```
int main(void)
{
    int num;

    printf("Enter a positive integer: ");
    scanf("%d", &num);

    printf("Sum of all natural numbers from %d to 1 = %d\n",
          num, addNumbers(num));

    return 0;
}
```

# Recursive Program to Sum Range of Natural Numbers

```
int addNumbers(int n)
{
    if (n != 0)
    {
        return n + addNumbers(n-1);
    }
    else
    {
        return n;
    }
}
```

Pass 1

n = 5

return 5 + Pass2(4)

Pass 2

n = 4

return 4 + Pass3(3)

Pass 3

n = 3

return 3 + Pass4(2)

Pass 4

n = 2

return 2 + Pass5(1)

Pass 5

n = 1

return 1 + Pass6(0)

Pass 6

n = 0


return 0

```
int main(void)
{
    int test = 0;
    printf("Enter a value ");
    scanf("%d", &test);

    int i;
    for (i = test; i > 0; i--)
        printf("%d ", i);
    for (i = 1; i <= test; i++)
        printf("%d ", i);

    return 0;
}
```

 frenchdm@omega:~

[frenchdm@omega ~]\$ 





What is the condition that makes it stop?

```
int main()
{
    int test = 0;
    printf("Enter a value ");
    scanf("%d", &test);

    int i;
    for (i = test; i > 0; i--)
        printf("%d ", i);
    for (i = 1; i <= test; i++)
        printf("%d ", i);

    return 0;
}
```

5	4	3	2	1	1	2	3	4	5
---	---	---	---	---	---	---	---	---	---

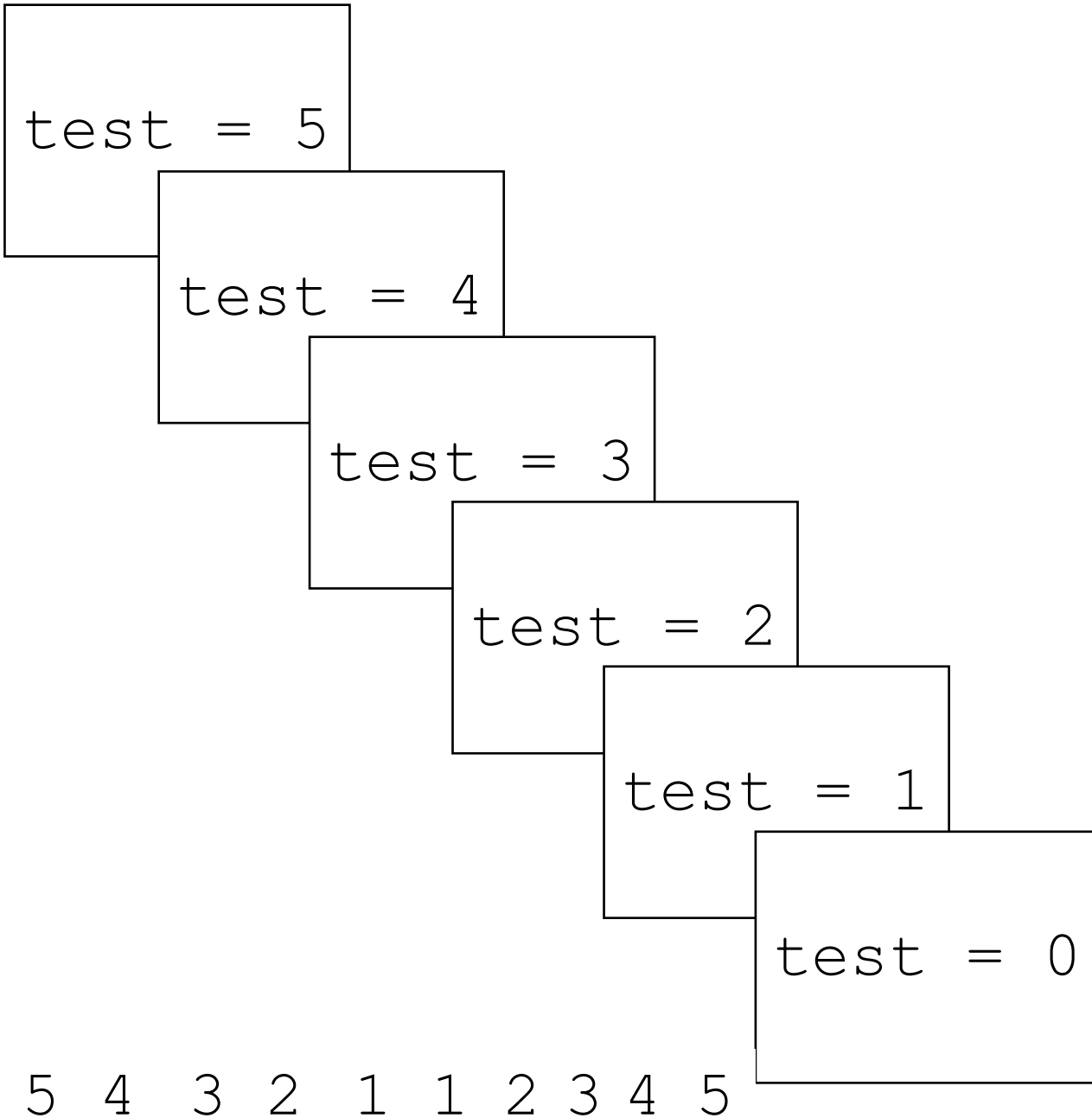
```
void printFun(int test)
{
    if (test < 1)
        return;

    else
    {
        printf("%d ", test);

        printFun(test-1);

        printf("%d ", test);

        return;
    }
}
```



```
void printFun(int test)
{
    if (test < 1)
        return;

    else
    {
        printf("%d ", test);
        printFun(test-1);
        printf("%d ", test);
        return;
    }
}
```

```
void printFun(int test)
{
    if (test < 1)
        return;

    else
    {
        printf("%d ", test);

        printFun(test-1);

        printf("%d ", test);

        return;
    }
}
```

```
void printFun(int test)
{
    if (test >= 1)
    {
        printf("%d ", test);

        printFun(test-1);

        printf("%d ", test);
    }
}
```

# Example Using Recursion: Fibonacci Series

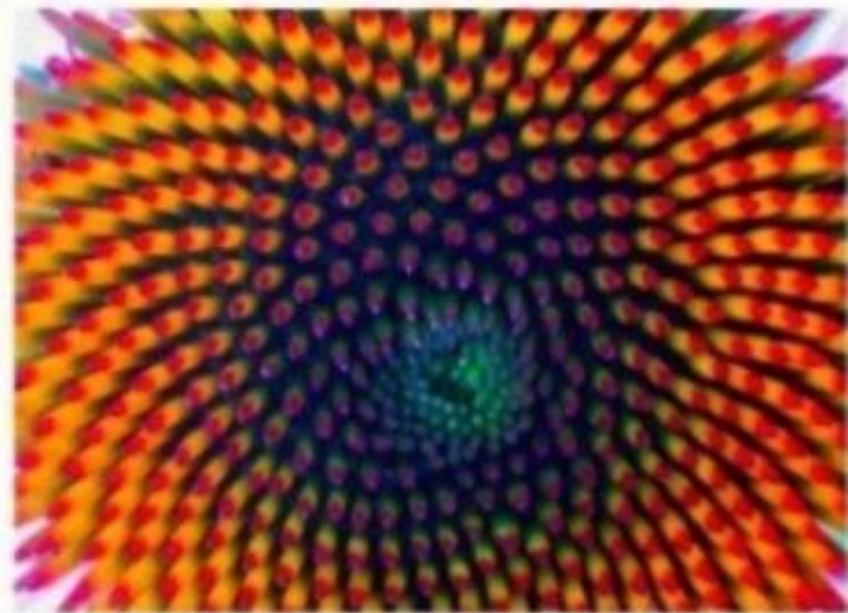
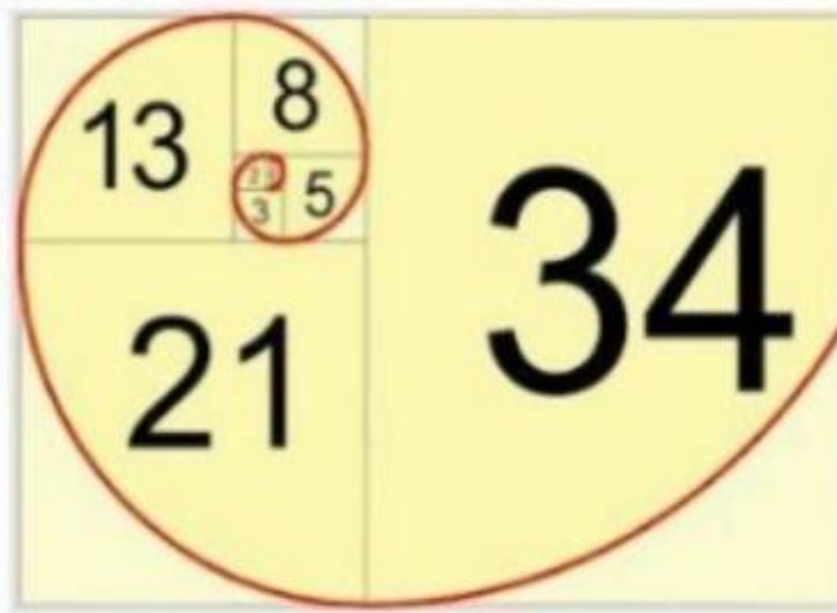
The Fibonacci series

0, 1, 1, 2, 3, 5, 8, 13, 21, ...

begins with 0 and 1 and has the property that each subsequent Fibonacci number is the sum of the previous two Fibonacci numbers.

The series occurs in nature and, in particular, describes a form of spiral.

The ratio of successive Fibonacci numbers converges to a constant value of 1.618....



# Example Using Recursion: Fibonacci Series

The Fibonacci series may be defined recursively as follows:

$$\text{fibonacci}(0) = 0$$

$$\text{fibonacci}(1) = 1$$

$$\text{fibonacci}(2) = 1$$

$$\text{fibonacci}(3) = 2$$

$$\text{fibonacci}(4) = 3$$

$$\text{fibonacci}(5) = 5$$

$$\text{fibonacci}(6) = 8$$

$$\text{fibonacci}(n) = \text{fibonacci}(n - 1) + \text{fibonacci}(n - 2)$$

## Example Using Recursion: Fibonacci Series

The Fibonacci series may be defined recursively as follows:

$$\text{fibonacci}(n) = \text{fibonacci}(n - 1) + \text{fibonacci}(n - 2)$$

We can create a program to calculate the  $n^{\text{th}}$  Fibonacci number recursively using a function we'll call `fibonacci`.

```
unsigned long long int result = fibonacci(number);
```

```
unsigned long long int fibonacci(unsigned int n)
```

```
{
```

```
    if (n == 0 || n == 1)
```

```
    {
```

```
        return n;
```

```
    }
```

```
else
```

```
{
```

```
    return fibonacci(n - 1) + fibonacci(n - 2);
```

```
}
```

```
}
```

```
Fibonacci(0) = 0
```

```
Fibonacci(1) = 1
```

```
Fibonacci(2) = 1
```

```
Fibonacci(3) = 2
```

```
Fibonacci(4) = 3
```

```
Fibonacci(5) = 5
```



```
#include <stdio.h>
```

```
void Hello(int x, int y, int z)
{
    if (z < 1 && x > 8)
        return;

    printf("%d%d%d\n", x, y, z);
    Hello(x+2, y/3, z-1);

    printf("%d%d%d\n", z, y, x);
}

int main()
{
    int x = 3, y = 4, z = 2;

    Hello(x, y, z);
}
```

342

511

700

007

115

243

```
#include <stdio.h>
```

```
void Hello(int x, int y, int z)
{
    if (z < 1 && x > 8)
        return;

    printf("%d%d%d\n", x, y, z);
    Hello(x+2, y/3, z-1);
    printf("%d%d%d\n", z, y, x);
}

int main()
{
    int x = 2, y = 4, z = 2;

    Hello(x, y, z);
}
```

242

411

600

80-1

-108

006

114

242

```

1 #include <stdio.h>
2
3 void PrintArray(int A[], int n)
4 {
5     int i = 0;
6     for (i = 0; i < n; i++)
7         printf("%d ", A[i]);
8
9     printf("\n");
10 }
11
12 int X(int A[], int n, int x, int i)
13 {
14     if (i == n)
15         return 1;
16
17     int z = A[i];
18
19     int y = X(A, n, x * A[i], i + 1);
20
21     A[i] = x * y;
22
23     PrintArray(A, n);
24
25     return z * y;
26 }
27
28 int main(void)
29 {
30     int A[] = {1, 2, 3, 4, 5};
31     int n = sizeof(A) / sizeof(A[0]);
32
33     X(A, n, 1, 0);
34
35     return 0;
36 }

```

From main -> X(A, 5, 1, 0)

1<sup>st</sup> call

n = 5, x = 1, i = 0

i != n (0 != 5)

z = A[i] = A[0] = 1

y = X(A, 5, 1, 1)

A[i] = A[0] = x \* y

PrintArray

return z\*y

2<sup>nd</sup> call

n = 5, x = 1, i = 1

i != n (1 != 5)

z = A[i] = A[1] = 2

y = X(A, 5, 2, 2)

A[i] = A[1] = x \* y

PrintArray

return z\*y

3<sup>rd</sup> call

n = 5, x = 2, i = 2

i != n (2 != 5)

z = A[i] = A[2] = 3

y = X(A, 5, 6, 3)

A[i] = A[2] = x \* y

PrintArray

return z\*y

4<sup>th</sup> call

n = 5, x = 6, i = 3

i != n (3 != 5)

z = A[i] = A[3] = 4

y = X(A, 5, 24, 4)

A[i] = A[3] = x \* y

PrintArray

return z\*y

5<sup>th</sup> call

n = 5, x = 24, i = 4

i != n (4 != 5)

z = A[i] = A[4] = 5

y = X(A, 5, 120, 5)

A[i] = A[4] = x \* y

PrintArray

return z\*y

6<sup>th</sup> call

n = 5, x = 120, i = 5

i != n (5 == 5)

return 1

3<sup>rd</sup> call

$n = 5, x = 2, i = 2$

$i \neq n$  ( $2 \neq 5$ )

$z = A[i] = A[2] = 3$

$y = X(A, 5, 6, 3)$

$A[i] = A[2] = x * y$

PrintArray

return  $z * y$

$y = 20$

$A[2] = x * y = 2 * 20$

{1, 2, 40, 30, 24}

return  $z * y$  ( $3 * 20$ )  $\Rightarrow 60$

5<sup>th</sup> call

$n = 5, x = 24, i = 4$

$i \neq n$  ( $4 \neq 5$ )

$z = A[i] = A[4] = 5$

$y = X(A, 5, 120, 5)$

$A[i] = A[4] = x * y$

PrintArray

return  $z * y$

$y = 1$

$A[4] = x * y = 24 * 1$

{1, 2, 3, 4, 24}

return  $z * y$  ( $5 * 1$ )  $\Rightarrow 5$

4<sup>th</sup> call

$n = 5, x = 6, i = 3$

$i \neq n$  ( $3 \neq 5$ )

$z = A[i] = A[3] = 4$

$y = X(A, 5, 24, 4)$

$A[i] = A[3] = x * y$

PrintArray

return  $z * y$

$y = 5$

$A[3] = x * y = 6 * 5$

{1, 2, 3, 30, 24}

return  $z * y$  ( $4 * 5$ )  $\Rightarrow 20$

6<sup>th</sup> call

$n = 5, x = 120, i = 5$

$i \neq n$  ( $5 == 5$ )

return 1

1<sup>st</sup> call

$n = 5, x = 1, i = 0$

$i \neq n$  ( $0 \neq 5$ )

$z = A[i] = A[0] = 1$

$y = X(A, 5, 1, 1)$

$A[i] = A[0] = x * y$

PrintArray

return  $z * y$

$y = 120$

$A[0] = x * y = 1 * 120$

{120, 60, 40, 30, 24}

return  $z * y$  ( $1 * 120$ ) => 120

2<sup>nd</sup> call

$n = 5, x = 1, i = 1$

$i \neq n$  ( $1 \neq 5$ )

$z = A[i] = A[1] = 2$

$y = X(A, 5, 2, 2)$

$A[i] = A[1] = x * y$

PrintArray

return  $z * y$

$y = 60$

$A[1] = x * y = 1 * 60$

{1, 60, 40, 30, 24}

return  $z * y$  ( $2 * 60$ ) => 120

main() does use the return value of the recursive function so the program ends.

The result of all printing is

1 2 3 4 24

1 2 3 30 24

1 2 40 30 24

1 60 40 30 24

120 60 40 30 24

```

3  #include <stdio.h>
4
5  int FunctionR(int z)
6  {
7      static int y = 0;
8
9      if(z != 0)
10     {
11         y++;
12         FunctionR(z/10);
13     }
14
15     return y;
16 }
17
18 int main(void)
19 {
20     int x = 12335;
21
22     printf("%d", FunctionR(x));
23
24     return 0;
25 }

```

main() call FunctionR with  
12335

FunctionR – Pass 1

z = 12335

y = 0

z != 0 so y = 1

Call FunctionR with 1233

return y

FunctionR – Pass 2

z = 1233

y = 1

z != 0 so y = 2

Call FunctionR with 123

return y

FunctionR – Pass 3

z = 123

y = 2

z != 0 so y = 3

Call FunctionR with 12

return y

FunctionR – Pass 4

z = 12

y = 3

z != 0 so y = 4

Call FunctionR with 1

return y

FunctionR – Pass 5

z = 1

y = 4

z != 0 so y = 5

Call FunctionR with 0

return y

FunctionR – Pass 6

z = 0

return y (5)

# Using Recursion

Any problem that can be solved recursively can also be solved iteratively.

A recursive approach is normally chosen in preference to an iterative approach when the recursive approach more naturally mirrors the problem and results in a program that's easier to understand and debug.

Another reason to choose a recursive solution is that an iterative solution may not be apparent.