CSE 3318

Week of 06/10/2024

Instructor : Donna French

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
```

Coding Assignment 1

If you are on a Mac, run the Omega version of `FileGenerator.e` on Omega and use FileZilla to move the files back to your Mac.

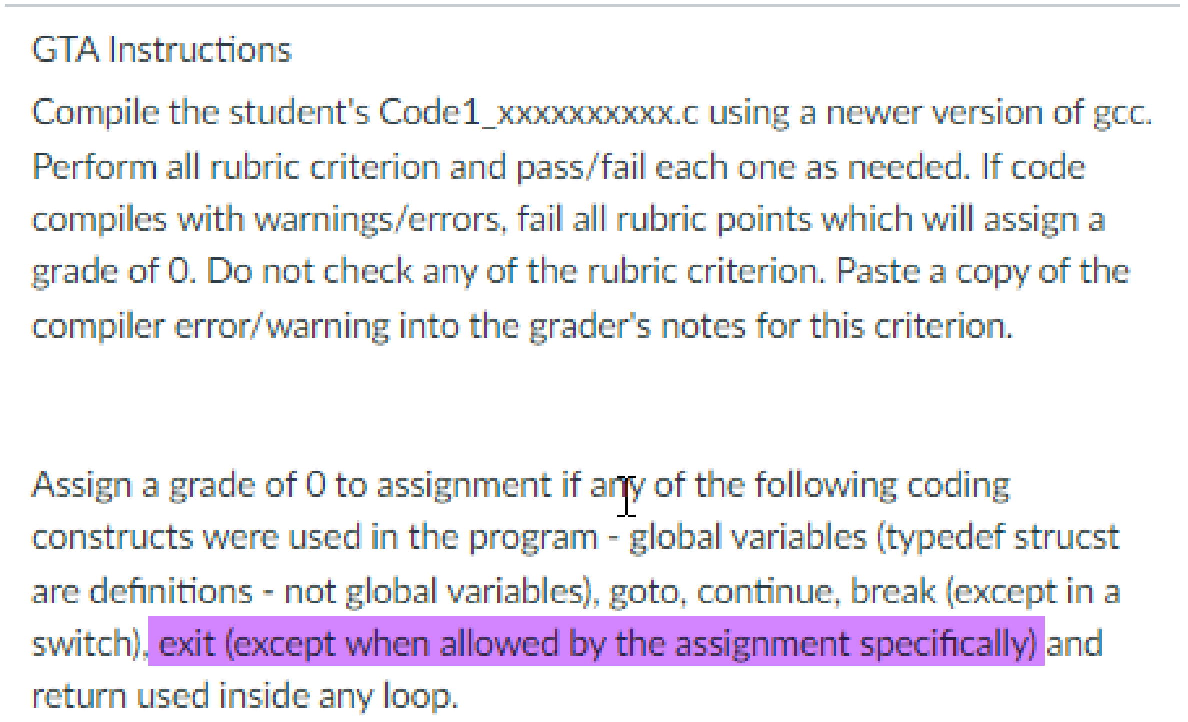
`FileGenerator.e` will NOT run on your Mac.

Create a function to read the file and call a function to add nodes to the linked list

Parameters : `argc`, `argv` and the linked list head

Return Value : `void`

Use `argv[1]` to get the filename from the command line if it is exists. If the program is not run with a file name on the command line, then print the message shown in the sample output and exit. If the program is run with a file name on the command line, then open the file with `"r"`. Use `fgets()` to loop through the file. See “File Handling in C” in the “Reviev Materials” module of Canvas. As each line of the file is read, call `AddNodeToLL()` to add the number read from the file as node in the linked list. Count how many records you read from the file/added to the linked list. Sum up the numbers read from the file/added to the linked list. Print the count and the sum to the screen - see sample output.



GTA Instructions

Compile the student's Code1_xxxxxxxxxx.c using a newer version of gcc. Perform all rubric criterion and pass/fail each one as needed. If code compiles with warnings/errors, fail all rubric points which will assign a grade of 0. Do not check any of the rubric criterion. Paste a copy of the compiler error/warning into the grader's notes for this criterion.

Assign a grade of 0 to assignment if any of the following coding constructs were used in the program - global variables (typedef strucst are definitions - not global variables), goto, continue, break (except in a switch), exit (except when allowed by the assignment specifically) and return used inside any loop.

Asymptotic Notation

But what happens as x grows?

$6x^2$

$6x^2 + 2x$

$6x^2 + 2x + 10$

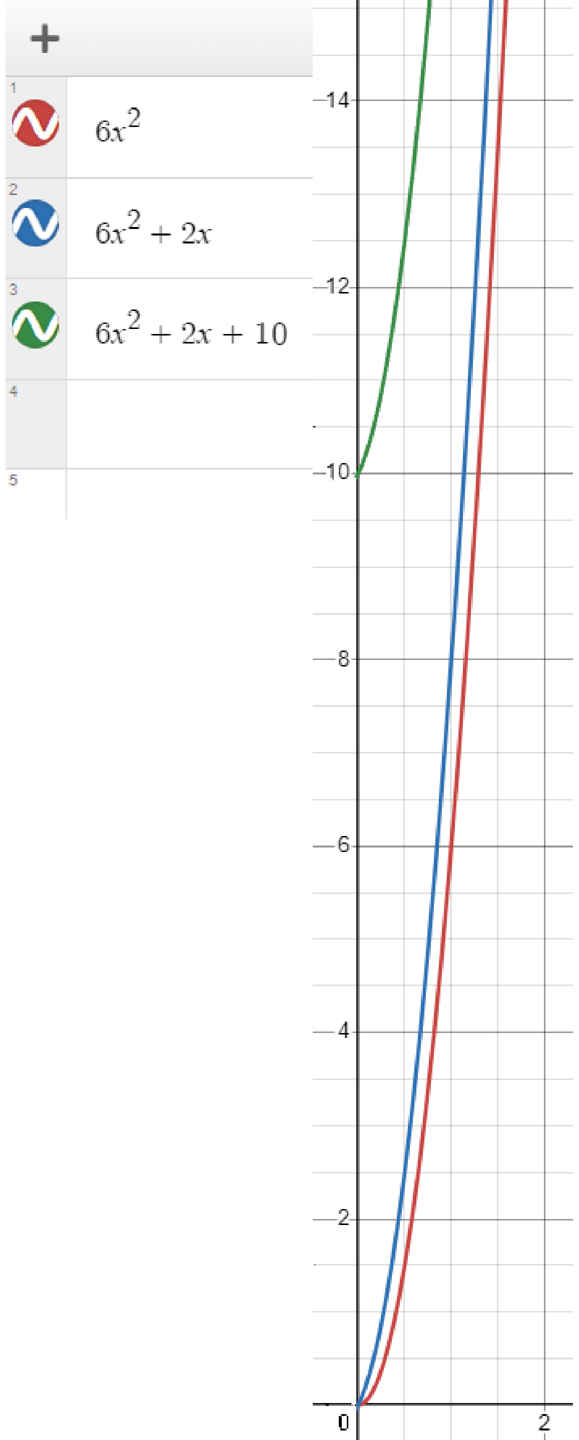
```
x = 5

6(5)²                    =     150
6(5)² + 2(5)             =     160
6(5)² + 2(5) + 10        =     170

x = 500

6(500)²                        =     1,500,000
6(500)² + 2(500)               =     1,501,000
6(500)² + 2(500) + 10          =     1,501,010

x = 1,000,000

6(1,000,000)²                              =     6,000,000,000,000
6(1,000,000)² + 2(1,000,000)               =     6,000,002,000,000
6(1,000,000)² + 2(1,000,000) + 10          =     6,000,002,000,010
```

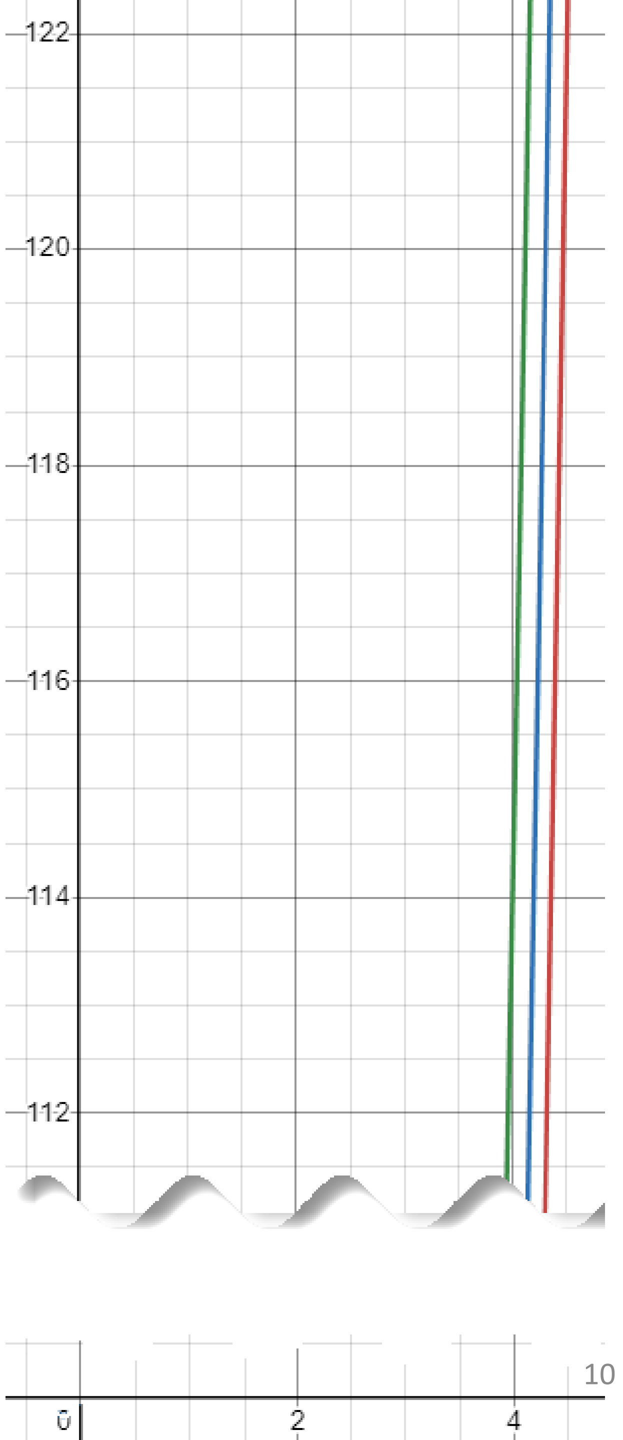
Asymptotic Notation

$6x^2$

$6x^2 + 2x$

$6x^2 + 2x + 10$





Asymptotic Notation

The larger x becomes, the less difference there is between the outcomes of the equations.

So if the running time of an algorithm is calculated as

$6n^2 + 2n + 100$

we can simplify the equation to just n^2 to describe the running time.

Dropping the coefficient 6 and the remaining terms ($2n + 100$) does not have a large enough impact on the overall running time as n approaches infinitely.

It doesn't really matter what coefficients we use - as long as the running time is $an^2 + bn + c$ for some numbers $a > 0$, b and c, there will always be a value of an^2 is greater than $bn + c$ and this difference increases as n increases.

Asymptotic Notation

By dropping the less significant terms and the constant coefficients, we can focus on the important part of an algorithm's running time

its rate of growth

without getting lost in details that could complicate our understanding and mislead us when comparing the running time of algorithms

When we drop the constant coefficients and the less significant terms, we use **asymptotic notation**.

We'll see three forms of it

- big-Θ notation (theta)
- big-O notation
- big-Ω notation (omega)

The definition of **asymptotic** is a line that approaches a curve but never touches.

A curve and a line that get closer but do not intersect are examples of a curve and a line that are **asymptotic** to each other.

Asymptotic Notation

Why is the coefficient usually not that important with algorithms?

Typically, we just want to compare the running times of algorithms that perform the same task.

> Algorithms tend to have different dominant terms (meaning they are in different complexity classes), which will allow us to easily identify which algorithms have faster running times for large values of *n*.

Calculating the coefficients for the running time of algorithms is often time consuming, error prone and inexact.

Identifying the most significant term for the running time is more straight forward.

Algorithms in the same complexity class that perform the same task typically have similar coefficients with some small differences that indicate improvements between algorithms in the same complexity class.

Asymptotic Notation

The words "typically" and "usually" and "similar" and "tend to" were used pretty heavily when describing why it's OK to drop coefficients and other terms.

We will soon see that the value of n plays a role is using these not-so-concrete descriptors.

We will study sorting algorithms (like insertion sort) that have a n^2 run time.

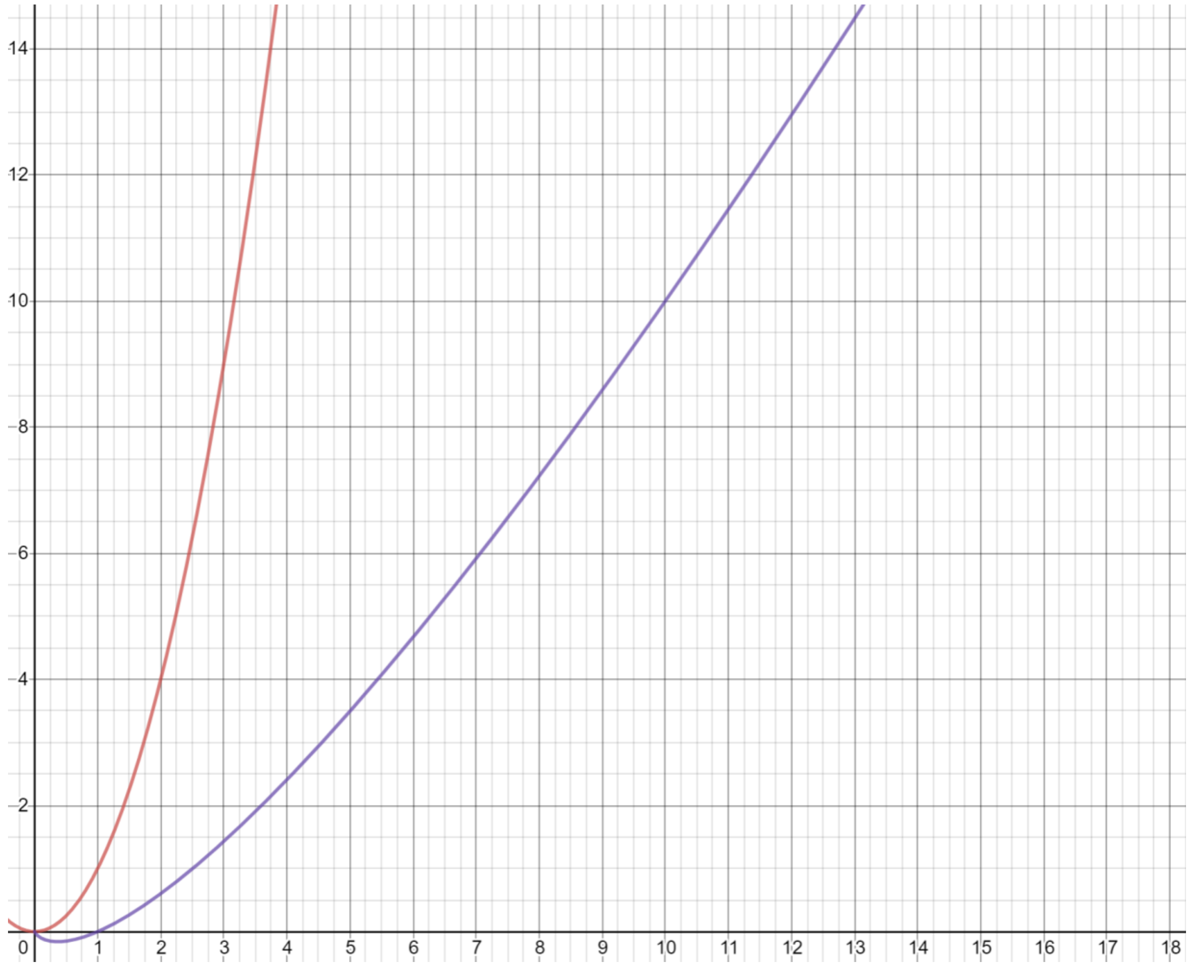
We will study sorting algorithms (like merge sort) have $n\log_2 n$ run times.

Asymptotic Notation

We have compared ***n^2*** to ***$n\log_2 n$*** run times.

We can see that ***n^2*** rises much quicker than ***$n\log_2 n$***, which, in general terms, means that ***$n\log_2 n$*** is the better algorithm in terms of running time.

So why would we ever use an algorithm with an ***n^2*** run time?



Asymptotic Notation

For smaller values of *n*, those coefficients can matter.

If the n^2 algorithms have small coefficients and the $n\log_2 n$ algorithms have large coefficients, then the coefficients will come into play when the value of *n* is smaller. Only with larger values of *n* do we see n^2 algorithms running slower than the $n\log_2 n$ algorithms.

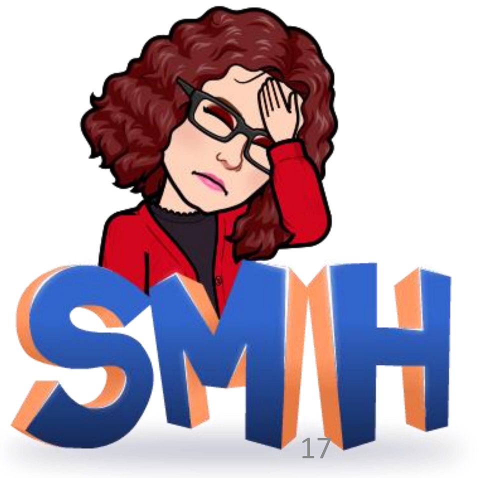
Asymptotic notation can be a really useful to talk about and compare algorithms.

It is definitely not without its limitations.

Asymptotic Notation

Let's look at a linear search function that returns when it finds the value

```
for (i = 0; i < array_size; i++)
{
    if (array[i] == SearchValue)
    {
        return i;
    }
}
return -1;
```



Asymptotic Notation

For each loop, several things happen

- check that `i < array_size`
- compare `array[i]` with `SearchValue`
- possibly return `i`
- increment `i`

```
for (i = 0; i < array_size; i++)
{
   if (array[i] == SearchValue)
   {
      return i;
   }
}
return -1;
```

Asymptotic Notation

Each step takes a constant amount of time each time the loop executes

- check that `i < array_size`
- compare `array[i]` with `SearchValue`
- possibly return `i`
- increment `i`

Let's call the sum of all those times c_1.

So if the for loop iterates n times, then the time needed for all iterations can be expressed as $c_1 n$

Asymptotic Notation

So what is the value of the c_1 in this c_1n formula?

The answer is

it depends

What is the

speed of the computer?

the programming language used?

the compiler or interpreter that translates the source program into runnable code?

other factors?

Asymptotic Notation

Are there steps in addition to the for loop steps?

`i` is initialized to 0

-1 will be returned when `SearchValue` is not found in the array

Let's sum up this time and call it c_2

```
for (i = 0; i < array_size; i++)
{
   if (array[i] == SearchValue)
   {
      return i;
   }
}
return -1;
```

Asymptotic Notation

So, the total time for linear search in the worst case is

$c_1 n + c_2$

We know the bigger n gets, the less significant c_1 and c_2 become.

So much so that we can drop them and just say the worst case is

n

where n is the size of the array to be searched.

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*

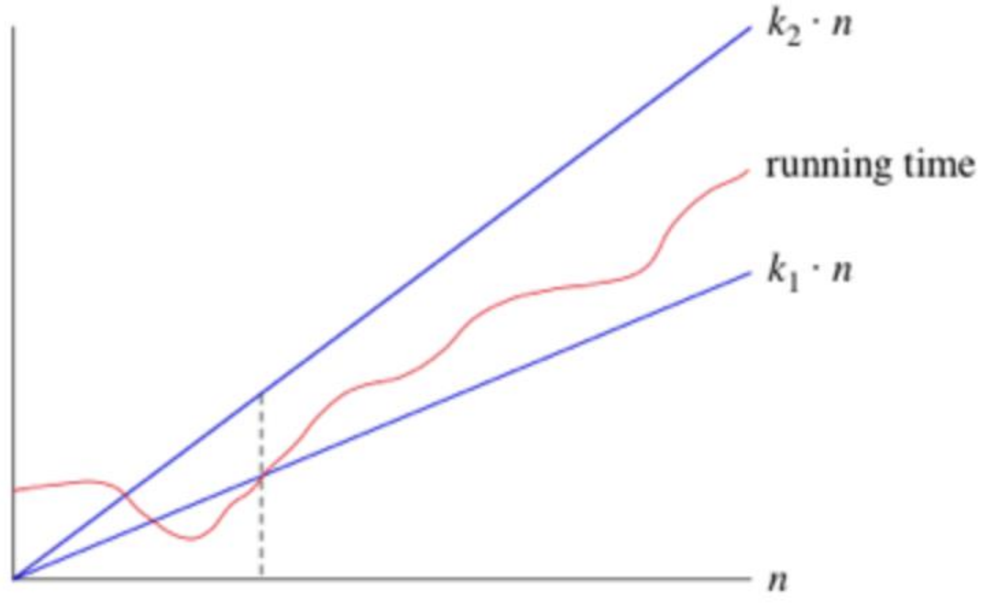
Asymptotic Notation

Θ(*n*)

When an algorithm's run time is described using Θ 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 Θ 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^{o} F or colder than -23^{o} F today – guaranteed!

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

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 Θ 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 Θ asymptotic notation from slowest to fastest in terms of growth

$\Theta(1)$

$\Theta(\log_2 n)$

$\Theta(n)$

$\Theta(n\log_2 n)$

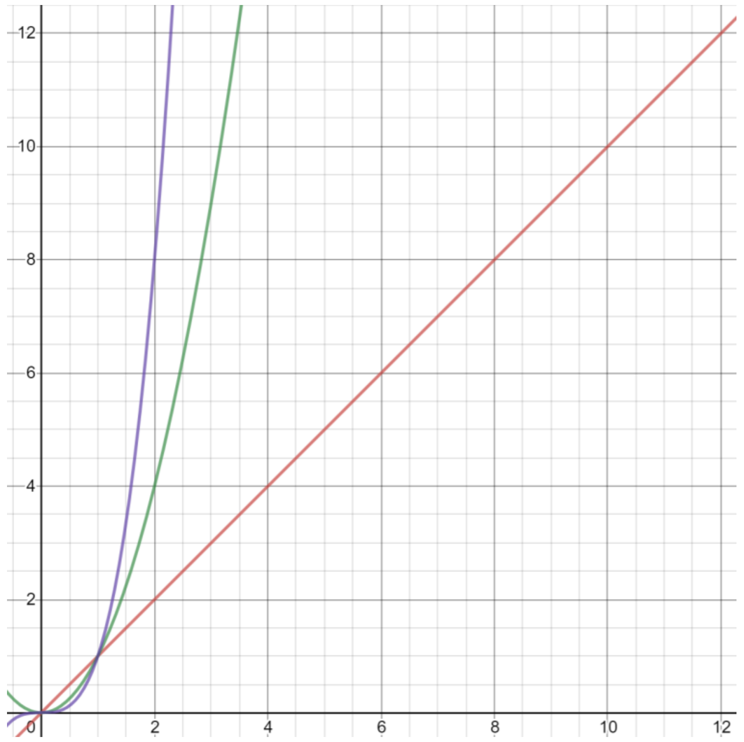
$\Theta(n^2)$

$\Theta(n^2\log_2 n)$

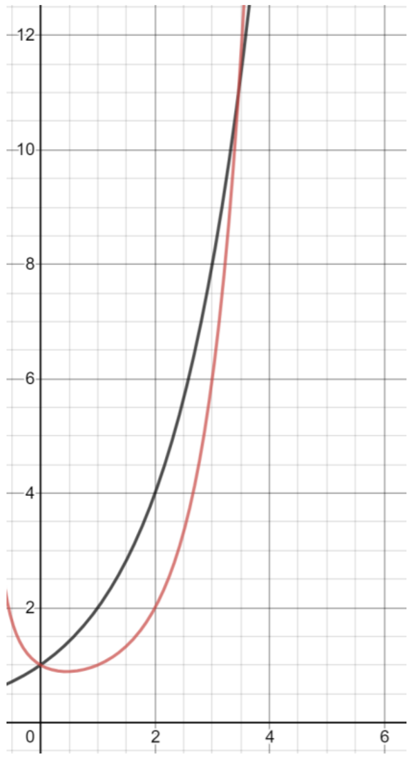
$\Theta(n^3)$

$\Theta(2^n)$

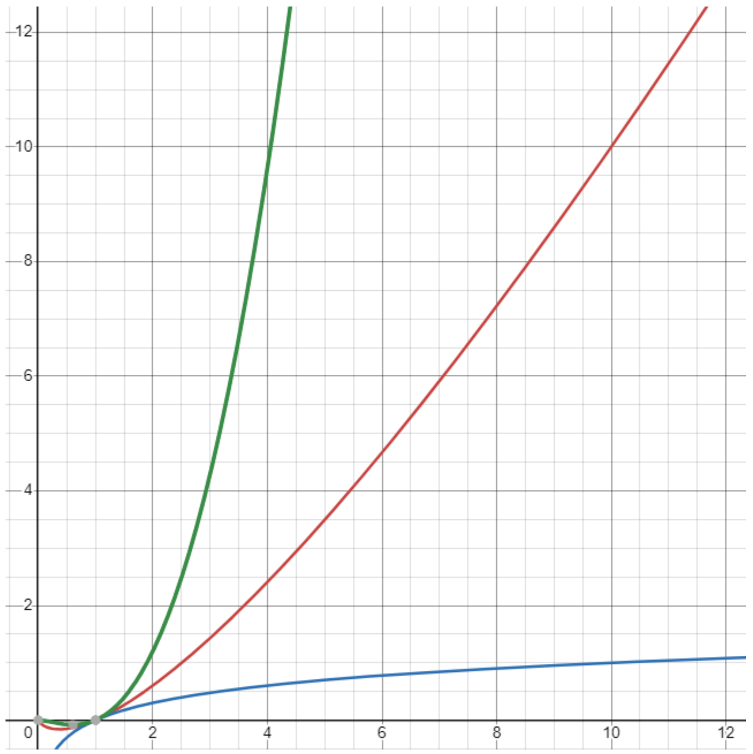
$\Theta(n!)$



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

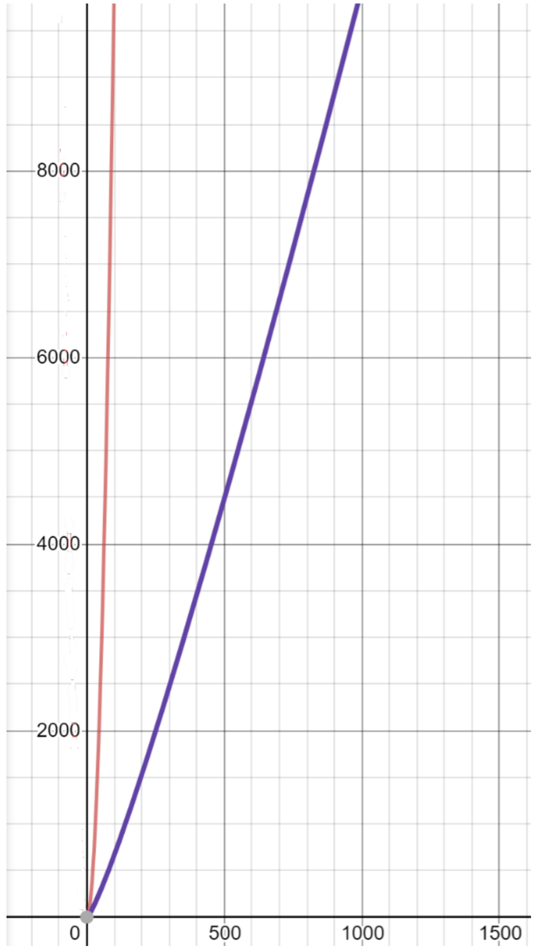
$$c_1 < c_2$$

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 ~33.

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

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

The Role of Algorithms in Computing

Insertion Sort $c_1 n^2$

$c_1 < c_2$

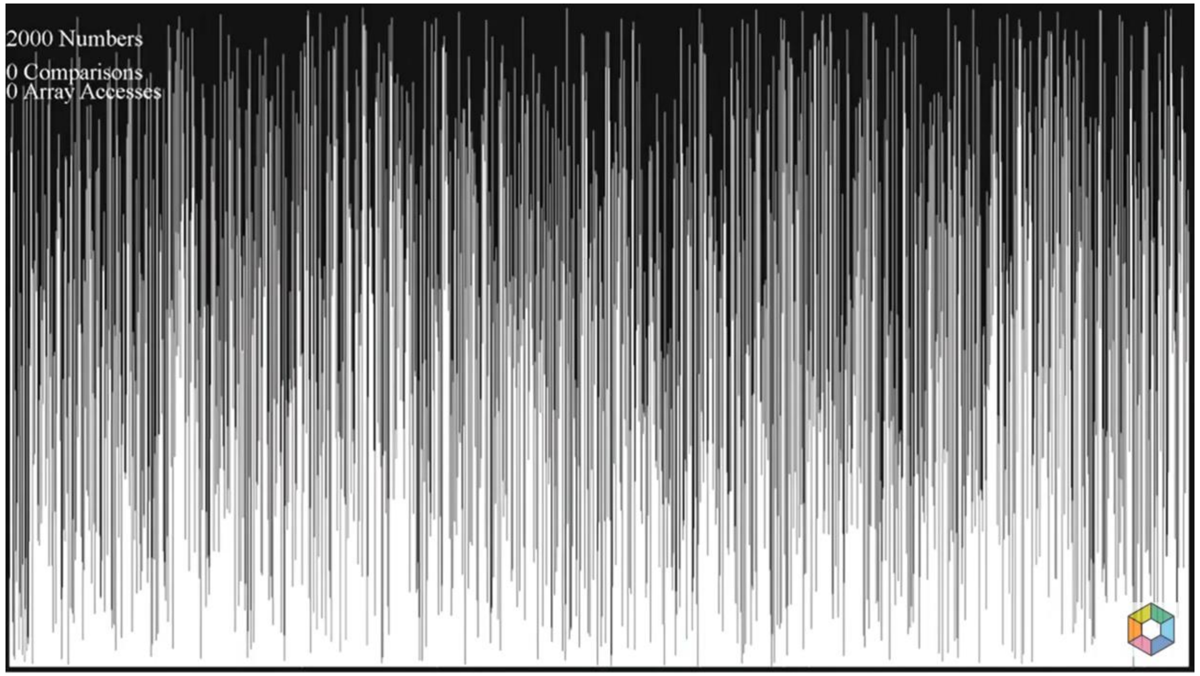
Merge Sort $c_2 n \log_2 n$

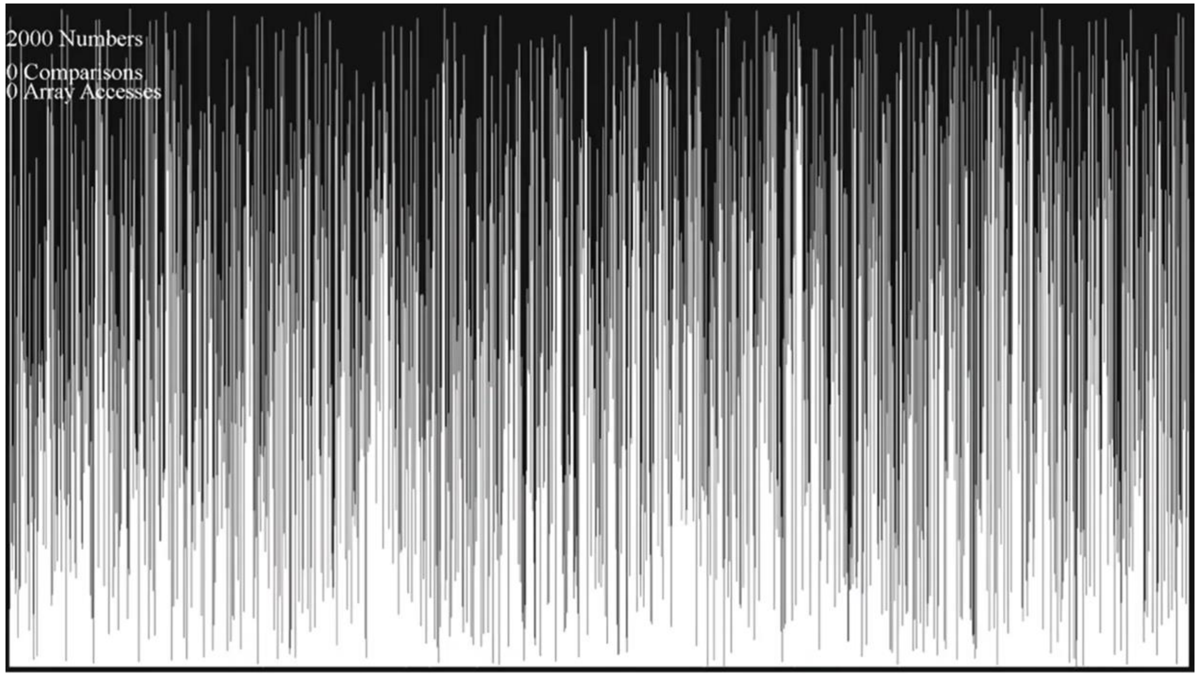
If n is 1,000,000, then n^2 is 1,000,000,000,000 and $n\log_2 n$ is ~200

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

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

$4000000c_1 = 2{,}893{,}035$ actions
$c_1 \approx 0.72$

$0.72n^2$

c_1 is less than c_2

Merge Sort $c_2 n \log_2 n$ n = 2000

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

$21940c_2 = 63{,}327$ actions
$c_2 \approx 2.89$

$2.89n \log_2 n$

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.

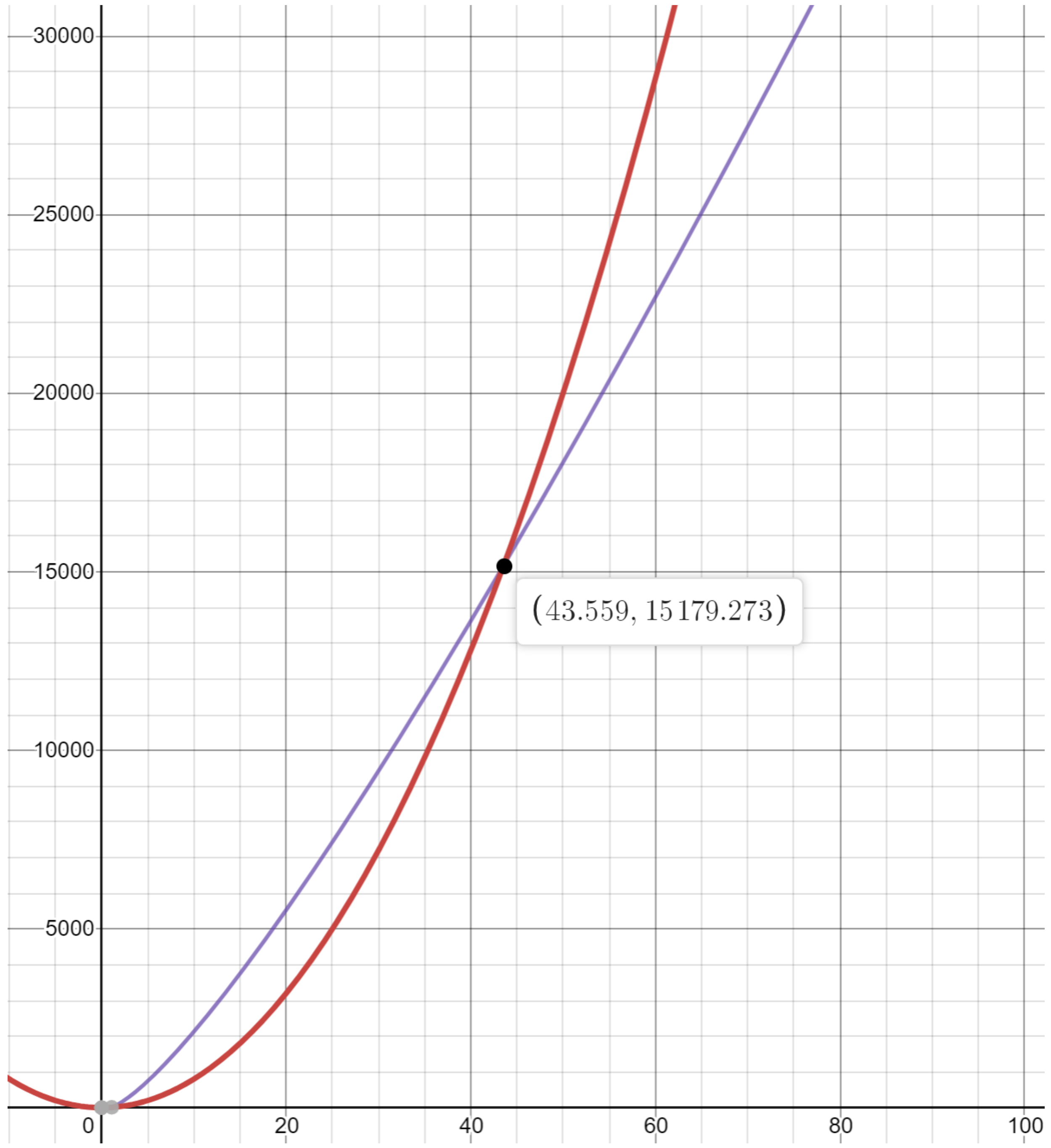
For which values of n does insertion sort beat merge sort?

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	**Merge**
$8n^2$	$64n\log_2 n$
n = 43	n = 43
$8(43)^2$	$64(43)(\log_2 43)$
14,792	~14,933

Insertion	**Merge**
$8n^2$	$64n\log_2 n$
n = 44	n = 44
$8(44)^2$	$64(44)(\log_2 44)$
15,488	~15,374

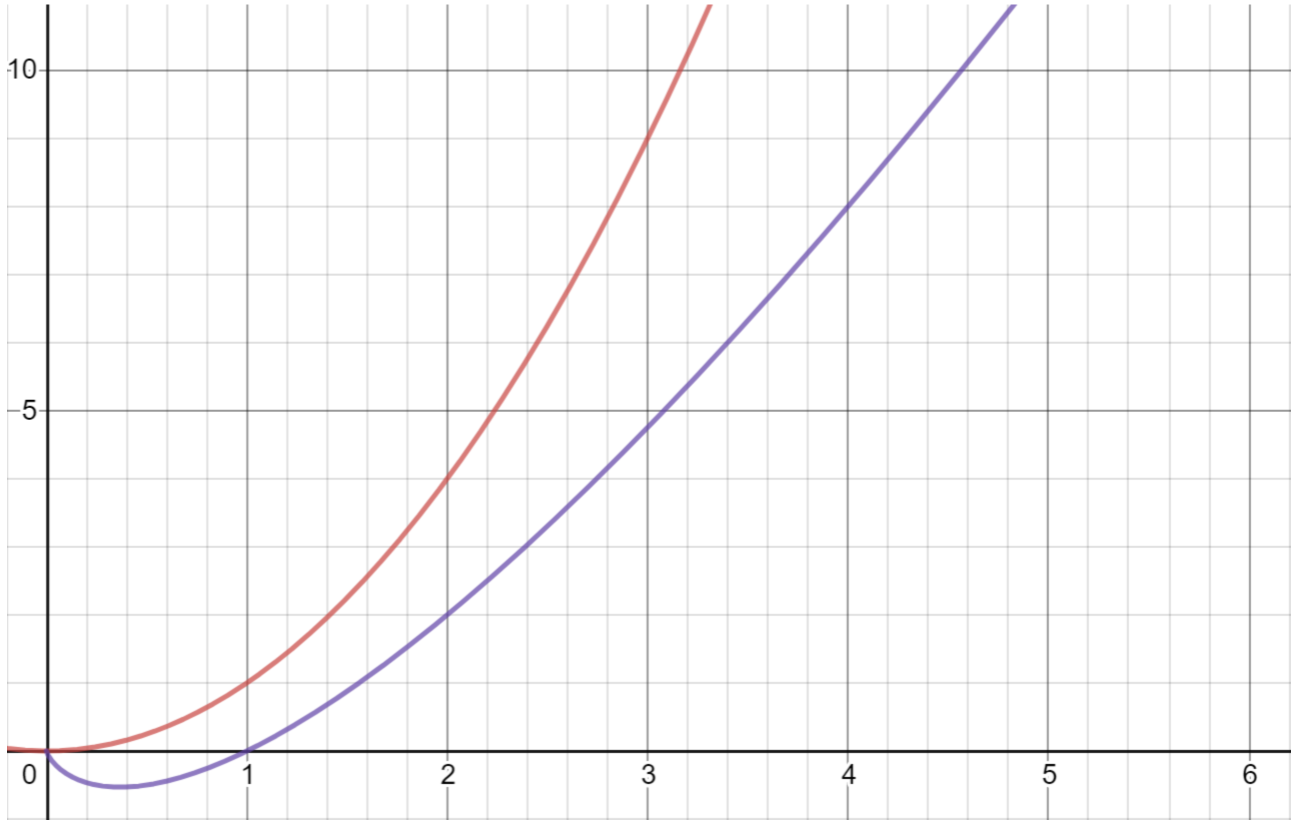
The Role of Algorithms in Computing

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 $\langle a_1, a_2,\ldots,an\rangle$.

Output : A permutation (reordering) $\langle a'1, a'2,\ldots a'_n\rangle$ of the input sequence such that $a'_1 \leq a'_2 \leq \ldots \leq a'_n$.

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

Insertion Sort

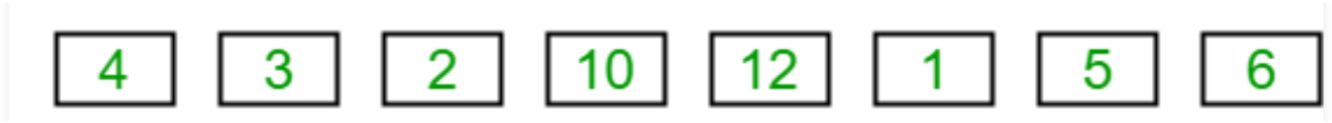
Insertion Sort algorithm sorts the input numbers in place.

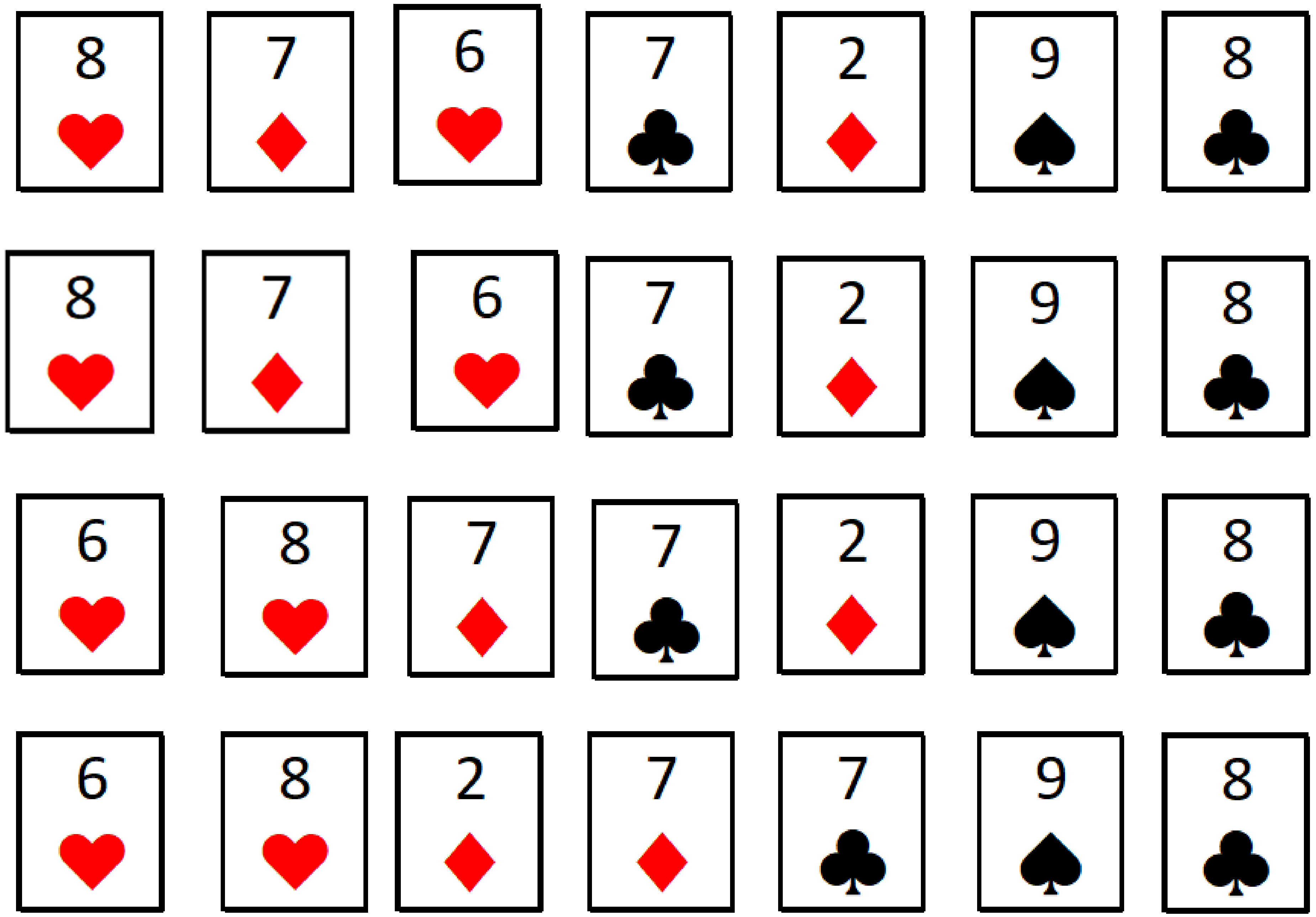
It rearranges the numbers within the array.

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




8♥ 7♦ 6♥ 7♣ 2♦ 9♠ 8♣
8♥ 7♦ 6♥ 7♣ 2♦ 9♠ 8♣
6♥ 8♥ 7♦ 7♣ 2♦ 9♠ 8♣
6♥ 8♥ 2♦ 7♦ 7♣ 9♠ 8♣

Insertion Sort

12, 11, 13, 5, 6

Why start at the 2nd element and not the first?

Let us loop for i = 1 (2nd 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

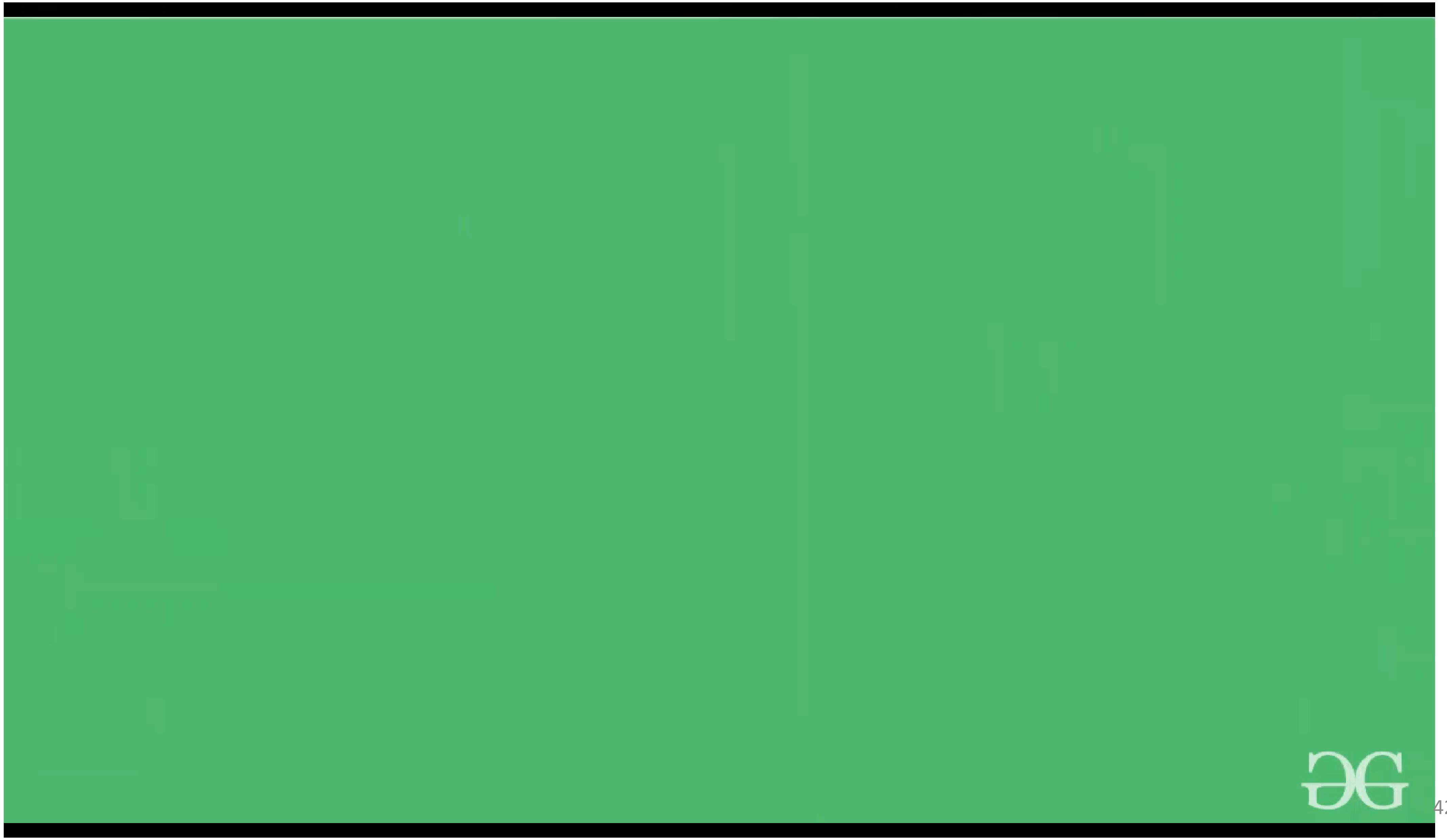
i = 3 5 will move to the beginning and all other elements from 11 to 13 will move one position ahead of their current position.

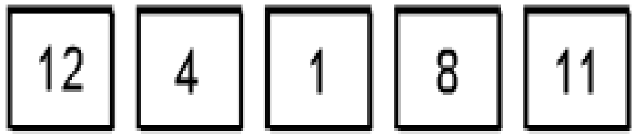
5, 11, 12, 13, 6

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





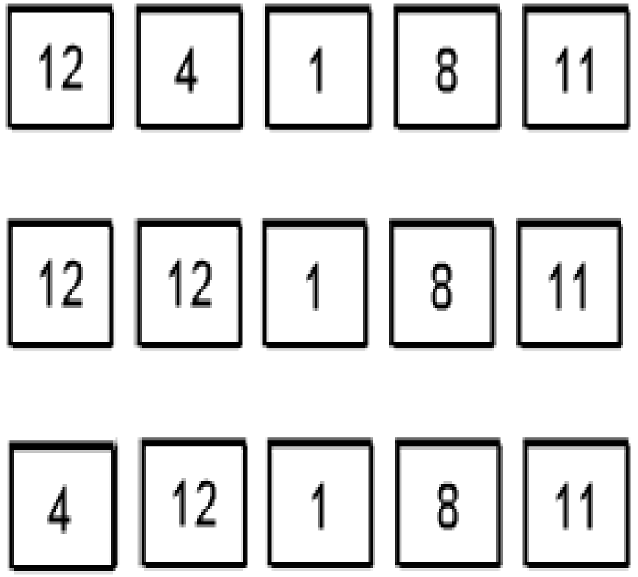
Insertion Sort

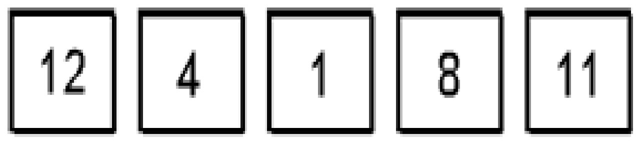
n is the number of elements to sort

```
for j = 2 to n

    key = A[j]
    // Insert A[j] into sorted sequence
    i = j - 1;

    while i > 0 and A[i] > key

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

    A[i + 1] = key;
```

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

1 2 3 4 5



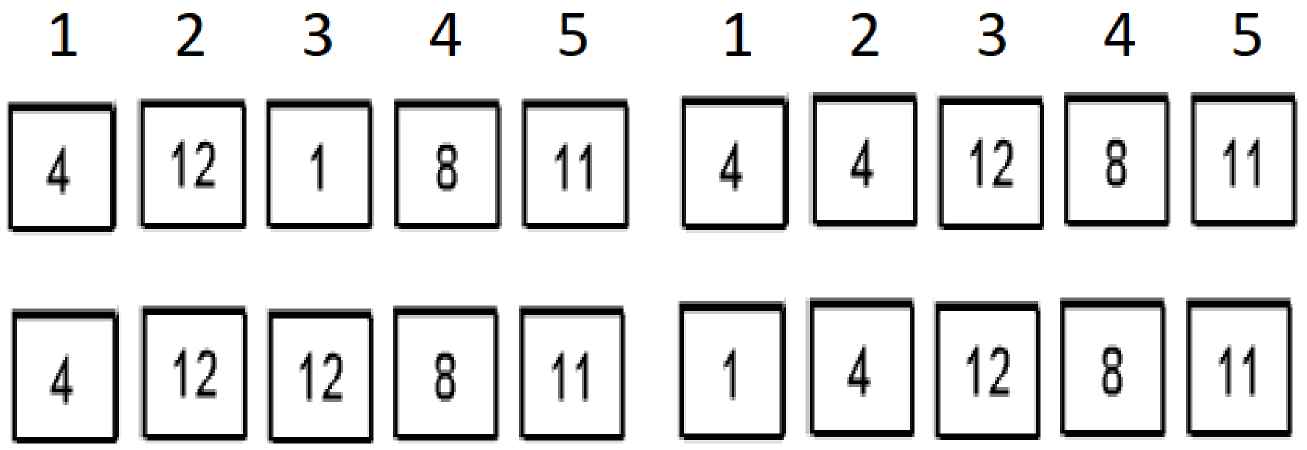


Insertion Sort

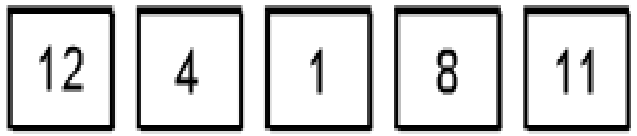
n is the number of elements to sort

```
for j = 2 to $n$

    key = A[j]
    // Insert A[j] into sorted sequence
    i = j - 1;

    while i > 0 and A[i] > key

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

    A[i + 1] = key;
```

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



n is the number of elements to sort

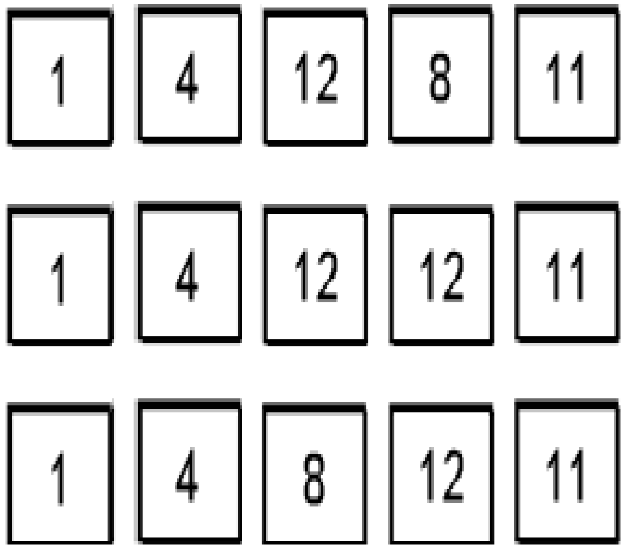


Insertion Sort

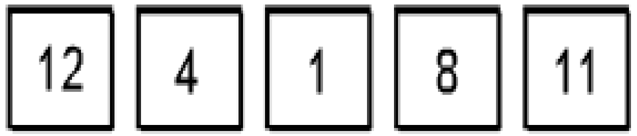
```
for j = 2 to n
    key = A[j]
    // Insert A[j] into sorted sequence
    i = j - 1;
    while i > 0 and A[i] > key
        A[i + 1] = A[i];
        i = i - 1;
    A[i + 1] = key;
```

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

1 2 3 4 5



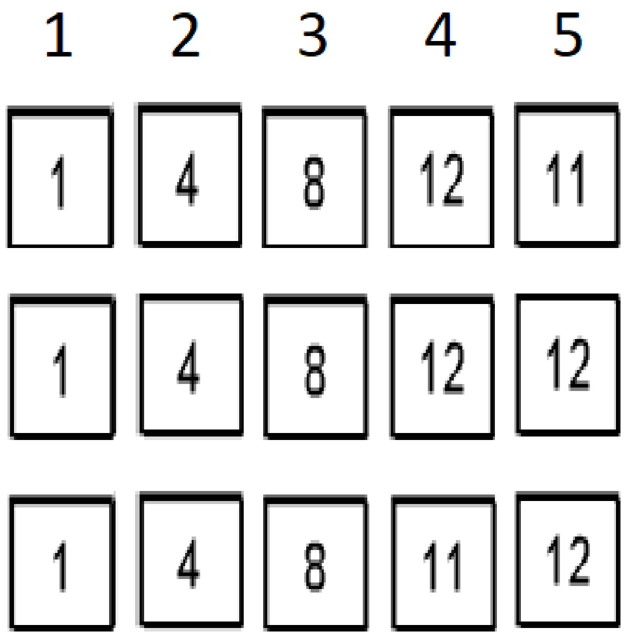
n is the number of elements to sort



Insertion Sort

```
for j = 2 to n

    key = A[j]

    // Insert A[j] into sorted sequence

    i = j - 1;

    while i > 0 and A[i] > key

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

        i = i - 1;

    A[i + 1] = key;
```

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



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
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	$\sum_{j=2}^{n} t_j$
6	A[i + 1] = A[i];	c_6	$\sum_{j=2}^{n}(t_j - 1)$
7	i = i - 1	c_7	$\sum_{j=2}^{n}(tj - 1)$
8	A[i + 1] = key	c_8	n - 1

Execute one less time than loop

Sum over the loop

Execute one less time than loop

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

$T(n) =$ $c_1 n + c_2(n - 1) + c_4(n - 1) +$

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

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

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

$c_8(n - 1)$

Analysis of Insertion Sort

Best Case Scenario

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

A[j] ≤ 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
      c6 and c7 did not execute
After  j=2 loop   5 6 11 12 13

Before j=3 loop   5 6 11 12 13
      c6 and c7 did not execute
After  j=3 loop   5 6 11 12 13

Before j=4 loop   5 6 11 12 13
      c6 and c7 did not execute
After  j=4 loop   5 6 11 12 13

Before j=5 loop   5 6 11 12 13
      c6 and c7 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

```
c1  for j = 2 to n
c2      key = A[j]
        // Insert A[j] into sorted part
c4      i = j - 1
c5      while i > 0 and A[i] > key
c6          A[i + 1] = A[i];
c7          i = i - 1
c8      A[i + 1] = key
```

Analysis of Insertion Sort

Best Case Scenario

Running time

$$
\begin{aligned}
T(n) = \quad & c_1 n + c_2(n-1) + c_4(n-1) + \\
& c_5 \sum_{j=2}^{n} t_j \quad + \\
& c_6 \sum_{j=2}^{n} (t_j - 1) + \\
& c_7 \sum_{j=2}^{n} (t_j - 1) + \\
& c_8(n-1)
\end{aligned}
$$

can be simplified to

$$T(n) = \quad c_1 n + c_2(n-1) + c_4(n-1) + c_5(n-1) + c_8(n-1)$$

Analysis of Insertion Sort

Best Case Scenario

Running time

$T(n) = \quad c_1 n + c_2(n-1) + c_4(n-1) + c_5(n-1) + c_8(n-1)$

can be further simplified to

$T(n) = \quad c_1 n + c_2 n - c_2 + c_4 n - c_4 + c_5 n - c_5 + c_8 n - c_8$

$c_1 n + c_2 n + c_4 n + c_5 n + c_8 n - c_2 - c_4 - c_5 - c_8$

$(c_1 + c_2 + c_4 + c_5 + c_8)n - (c_2 + c_4 + c_5 + c_8)$

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

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

Worst Case 13,12,11,6,5

Before j=2 loop 13 12 11 6 5 — c_1, c_2, c_4, c_8

i before 1 and i after 0 — c_5, c_6, c_7, c_5

After j=2 loop 12 13 11 6 5

Before j=3 loop 12 13 11 6 5 — c_1, c_2, c_4, c_8

i before 2 and i after 1 — c_5, c_6, c_7

i before 1 and i after 0 — c_5, c_6, c_7, c_5

After j=3 loop 11 12 13 6 5

Before j=4 loop 11 12 13 6 5 — c_1, c_2, c_4, c_8

i before 3 and i after 2 — c_5, c_6, c_7

i before 2 and i after 1 — c_5, c_6, c_7

i before 1 and i after 0 — c_5, c_6, c_7, c_5

After j=4 loop 6 11 12 13 5

Before j=5 loop 6 11 12 13 5 — c_1, c_2, c_4, c_8

i before 4 and i after 3 — c_5, c_6, c_7

i before 3 and i after 2 — c_5, c_6, c_7

i before 2 and i after 1 — c_5, c_6, c_7

i before 1 and i after 0 — c_5, c_6, c_7, c_5

After j=5 loop 5 6 11 12 13

Final value of j is 6 — c_1

```
c1  for j = 2 to n
c2      key = A[j]
        // Insert A[j] into sorted part
c4      i = j - 1
c5      while i > 0 and A[i] > key
c6          A[i + 1] = A[i];
c7          i = i - 1
c8      A[i + 1] = key
```

Before j=2 loop 13 12 11 6 5	c_1,c_2,c_4,c_8
i before 1 and i after 0	c_5,c_6,c_7,c_5
After j=2 loop 12 13 11 6 5	
Before j=3 loop 12 13 11 6 5	c_1,c_2,c_4,c_8
i before 2 and i after 1	c_5,c_6,c_7
i before 1 and i after 0	c_5,c_6,c_7,c_5
After j=3 loop 11 12 13 6 5	
Before j=4 loop 11 12 13 6 5	c_1,c_2,c_4,c_8
i before 3 and i after 2	c_5,c_6,c_7
i before 2 and i after 1	c_5,c_6,c_7
i before 1 and i after 0	c_5,c_6,c_7,c_5
After j=4 loop 6 11 12 13 5	
Before j=5 loop 6 11 12 13 5	c_1,c_2,c_4,c_8
i before 4 and i after 3	c_5,c_6,c_7
i before 3 and i after 2	c_5,c_6,c_7
i before 2 and i after 1	c_5,c_6,c_7
i before 1 and i after 0	c_5,c_6,c_7,c_5
After j=5 loop 5 6 11 12 13	
Final value of j is 6	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

t_j is the number of times the while loop runs - excludes check to fail

Analysis of Insertion Sort

Worst Case Scenario

Running time

$$T(n) = c_1 n + c_2(n-1) + c_4(n-1) + c_5\sum_{j=2}^{n} t_j + c_6\sum_{j=2}^{n}(t_j - 1) + c_7\sum_{j=2}^{n}(t_j - 1) + c_8(n-1)$$

c_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

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

t_j is the number of times the while loop runs - excludes check to fail

Analysis of Insertion Sort

Worst Case Scenario

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

therefore,

t_j + 1 more pass to fail = *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

t_j is the number of times the while loop runs - excludes check to fail

Analysis of Insertion Sort

Worst Case Scenario

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

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

c_5

14 times
2+3+4+5

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

t_j is the number of times the while loop runs - excludes check to fail

Analysis of Insertion Sort

Worst Case Scenario

$\frac{n(n-1)}{2}$ 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

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

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

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

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

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

t_j is the number of times the while loop runs - excludes check to fail

Analysis of Insertion Sort

Worst Case Scenario

Running time

$T(n) = c_1 n + c_2(n-1) + c_4(n-1) +$
$c_5 \sum_{j=1}^{n} j \quad - 1 \;+$
$c_6 \sum_{j=2}^{n} (t_j$ - 1) +
$c_7 \sum_{j=2}^{n} (t_j$ - 1) +
$c_8(n-1)$

$T(n) = c_1 n + c_2(n-1) + c_4(n-1) +$
$c_5(\frac{n(n+1)}{2}$ - 1) +
$c_6(\frac{n(n-1)}{2}) +$
$c_7\,(\frac{n(n-1)}{2}) +$
$c_8(n-1)$

t_j is the number of times the while loop runs - excludes check to fail

Analysis of Insertion Sort

Worst Case Scenario

Running time

$$T(n) = c_1 n + c_2(n-1) + c_4(n-1) + c_5\left(\frac{n(n+1)}{2} - 1\right) + c_6\left(\frac{n(n-1)}{2}\right) + c_7\left(\frac{n(n-1)}{2}\right) + c_8(n-1)$$

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

t_j is the number of times the while loop runs - excludes check to fail

Analysis of Insertion Sort

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.

Asymptotic Notation

We use big-Θ 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.

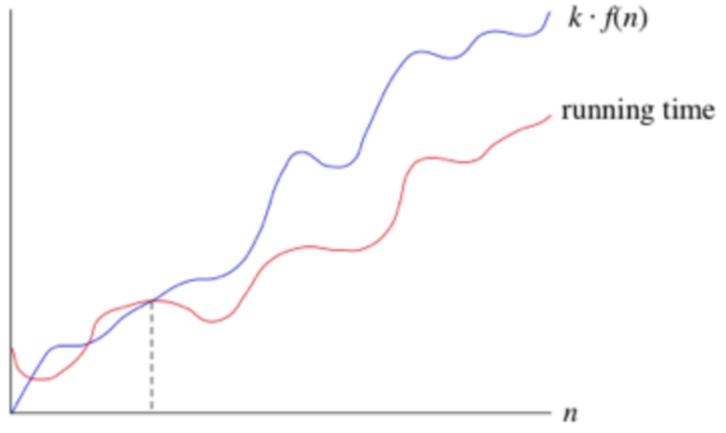
The running time of binary search is never worse than $\Theta(\log_2 n)$, but it's sometimes better.

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

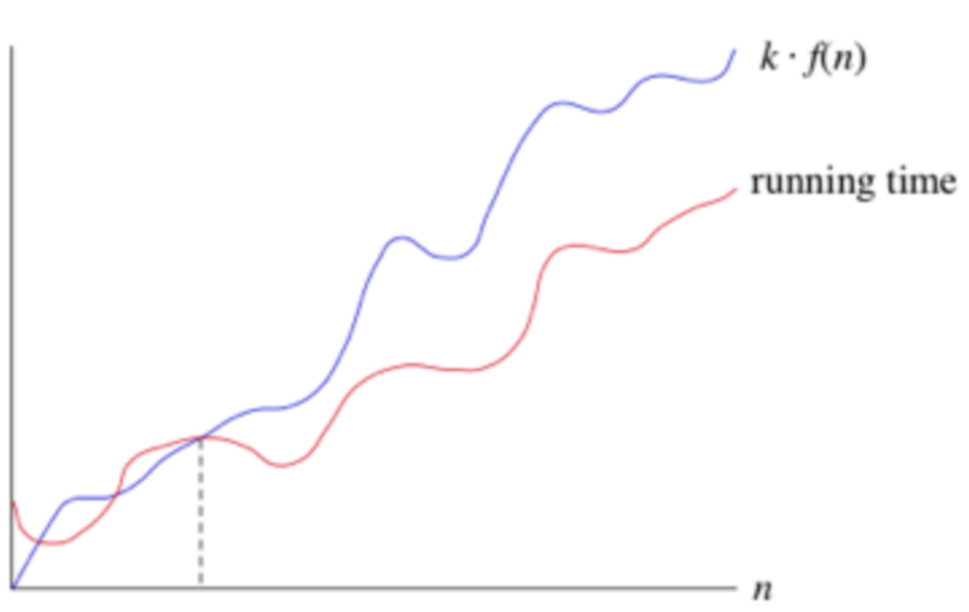
Asymptotic Notation

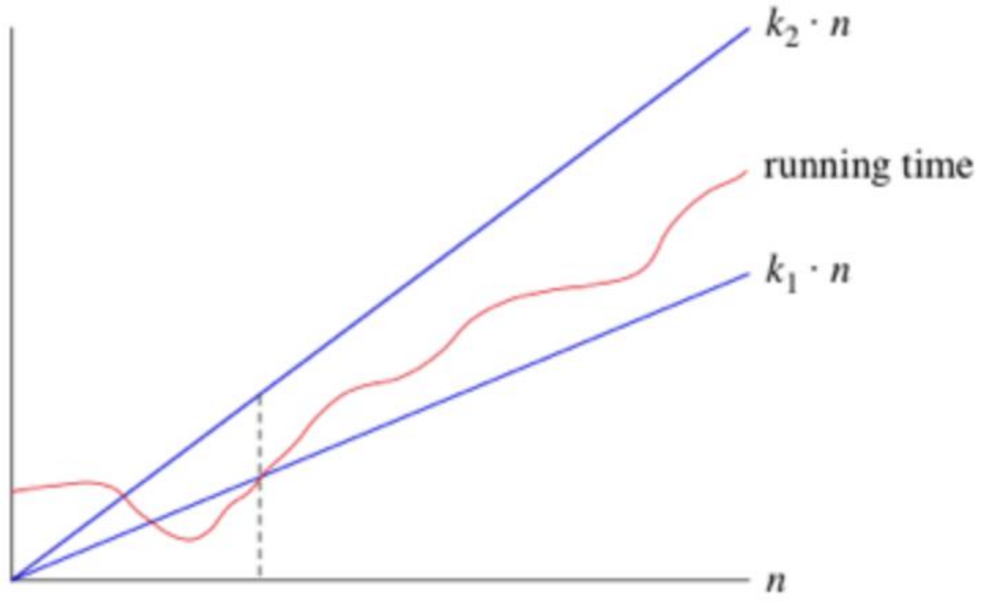
"big-O" notation means "the running time grows at most this much, but it could grow more slowly."

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 Θ
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

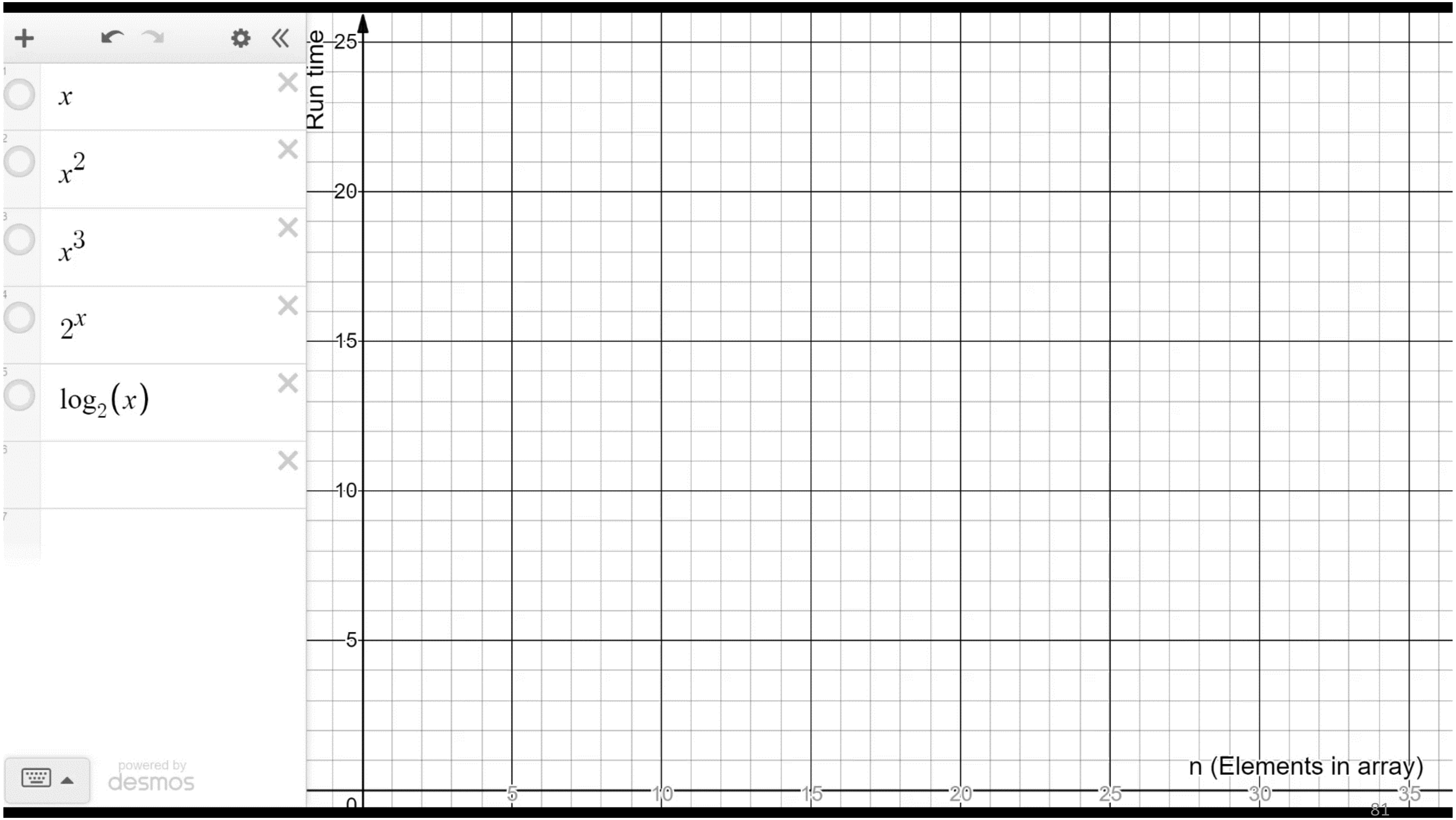
The worst-case running time of binary search is $\Theta(\log_2 n)$ time but it would also be accurate (not precise) to state that binary search has a run time of $O(n)$, $O(n^2)$, $O(n^3)$, and $O(2^n)$.

For example, if n = 4…

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

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.



1 x

2 x^2

3 x^3

4 2^x

5 $\log_2(x)$

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

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.

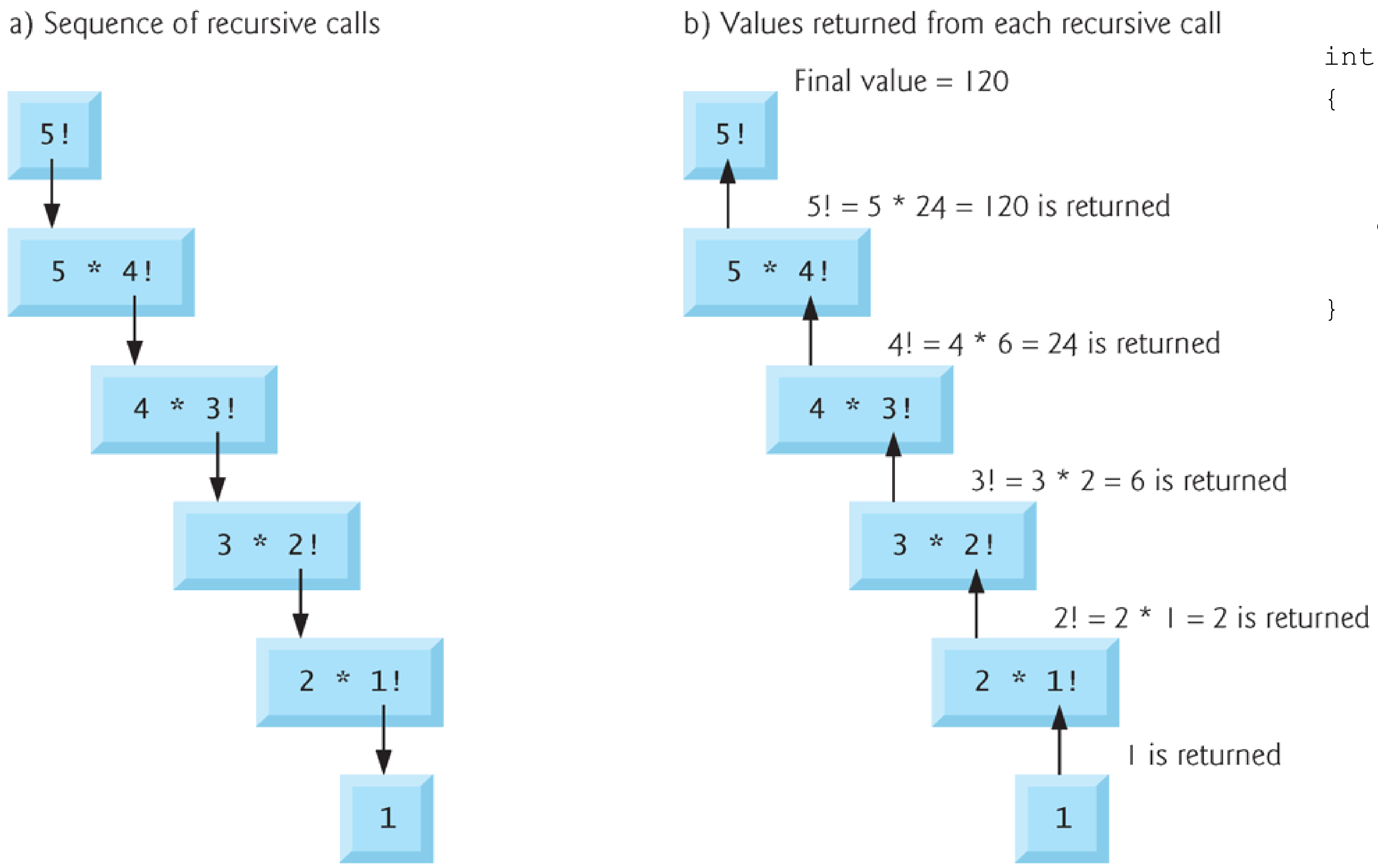
A simple example of a mathematical recursion is factorial

```
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
5!
5 * 4!
4 * 3!
3 * 2!
2 * 1!
1
b) Values returned from each recursive call
Final value = 120
5!
5! = 5 * 24 = 120 is returned
5 * 4!
4! = 4 * 6 = 24 is returned
4 * 3!
3! = 3 * 2 = 6 is returned
3 * 2!
2! = 2 * 1 = 2 is returned
2 * 1!
1 is returned
1

```
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
```

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

```
Enter 4

Calls factorial with 4

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.

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:~

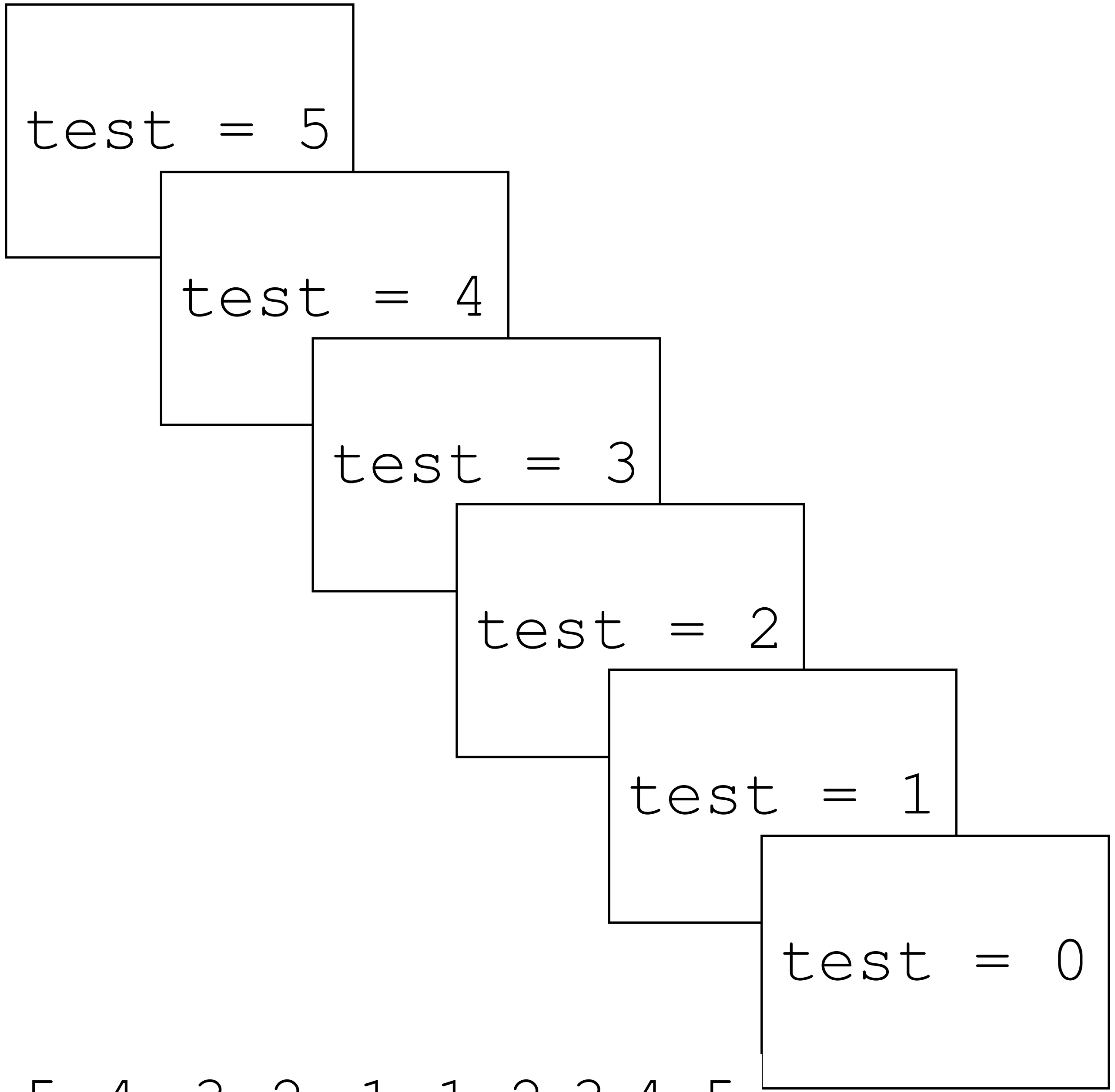
```
[frenchdm@omega ~]$ 
```

What is the condition that makes it stop?

```
int ma
{
   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;
   }
}
```



```
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;
  }
}
```

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

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

    printFun(test-1);

    printf("%d ", test);

    return;
   }
}
```

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

    printFun(test-1);

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

Example Using Recursion: Fibonacci Series

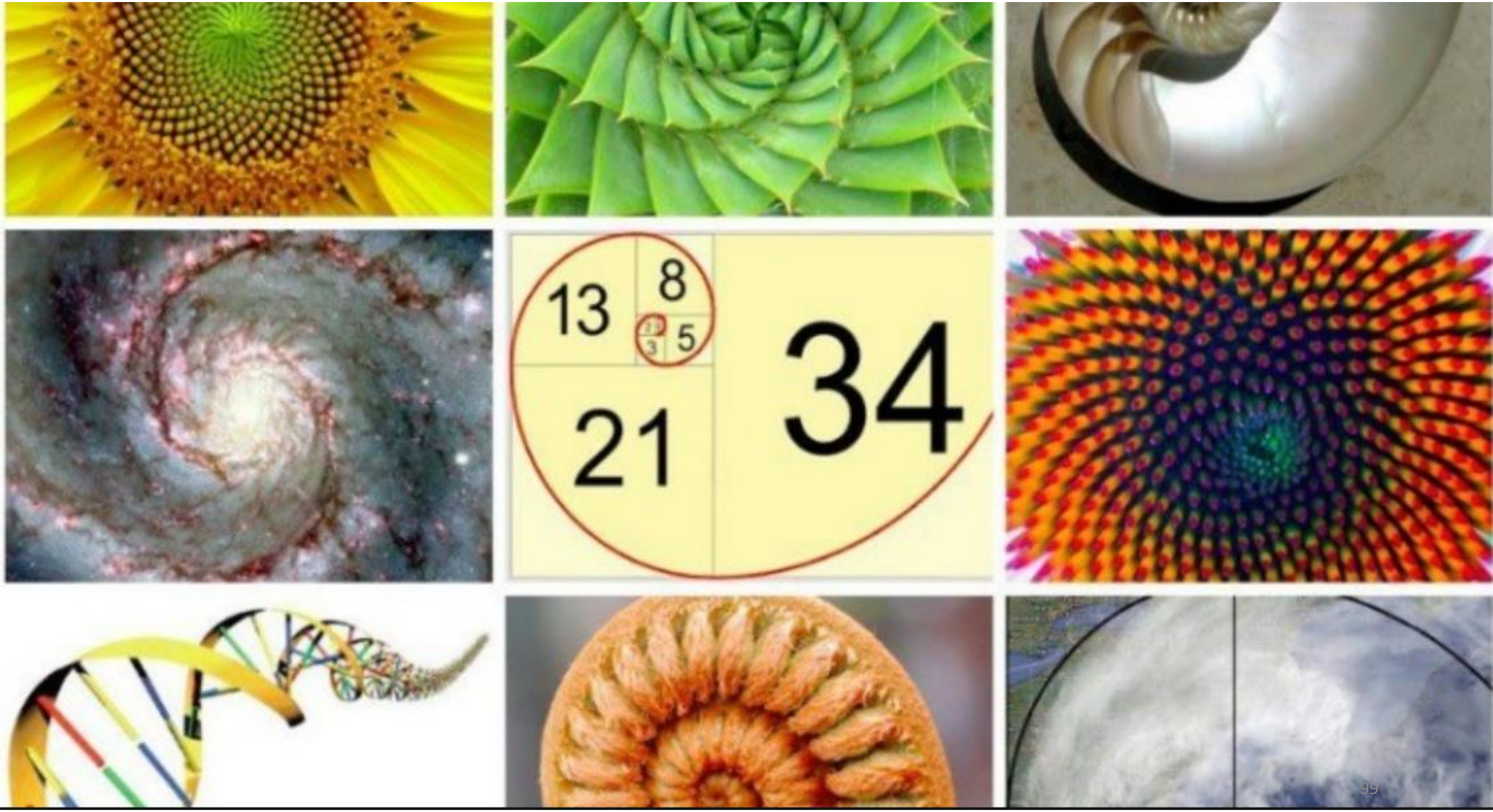
The Fibonacci series

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….


13
8
3
5
34
21

Example Using Recursion: Fibonacci Series

The Fibonacci series may be defined recursively as follows:

fibonacci(0) = 0
fibonacci(1) = 1
fibonacci(2) = 1
fibonacci(3) = 2
fibonacci(4) = 3
fibonacci(5) = 5
fibonacci(6) = 8

fibonacci(*n*) *= fibonacci(n – 1) + fibonacci(n – 2)*

Example Using Recursion: Fibonacci Series

The Fibonacci series may be defined recursively as follows:

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

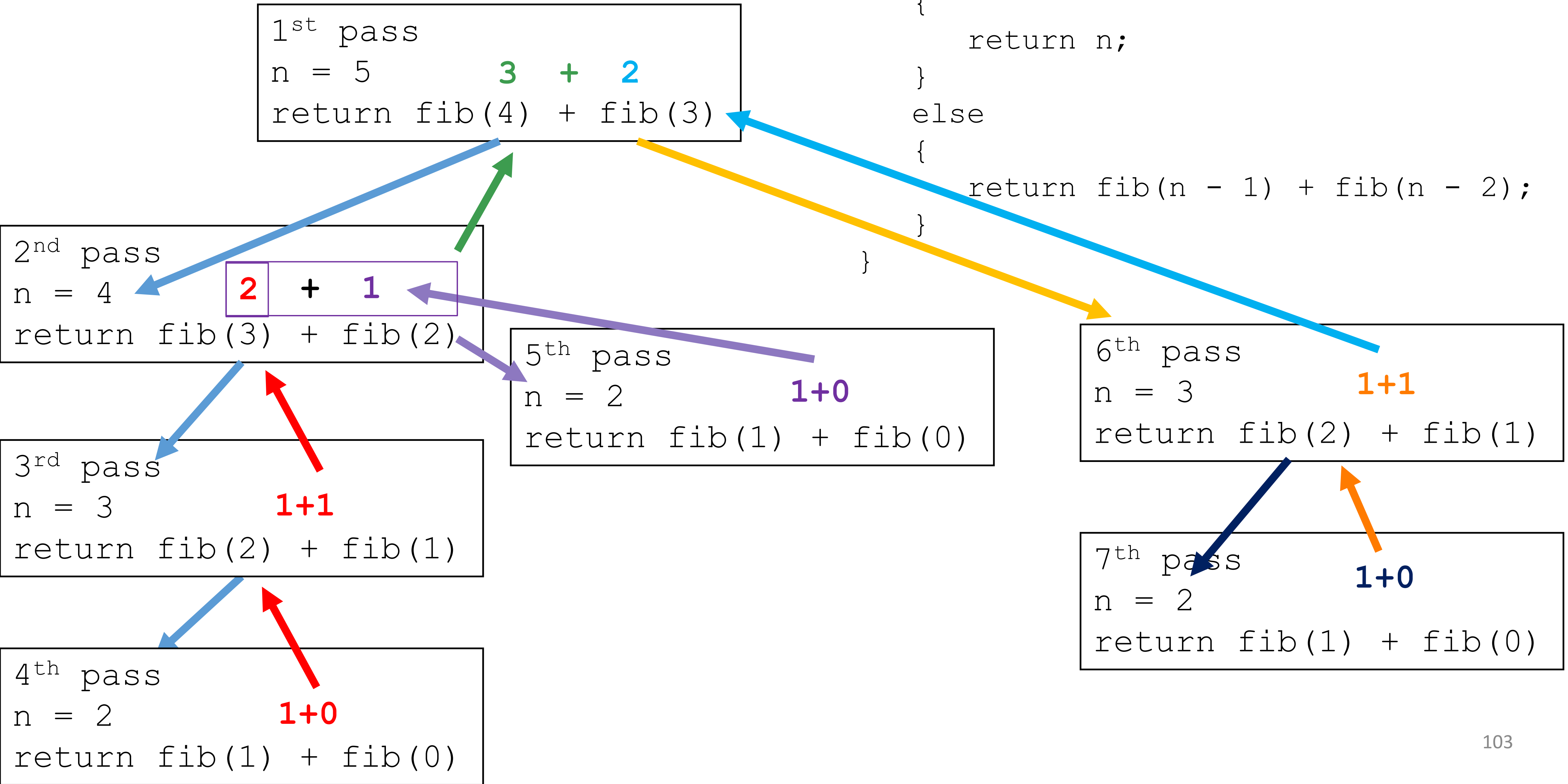
We can create a program to calculate the *n*th Fibonacci number recursively using a function we'll call `fibonacci`.

```
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
```

fib(5)

```
unsigned long long int fib(unsigned int n)
{
    if (n == 0 || n == 1)
    {
        return n;
    }
    else
    {
        return fib(n - 1) + fib(n - 2);
    }
}
```



```c
// Recursive power function
#include <stdio.h>

long power(long base, long exponent)
{
    if (exponent == 1)
        return base;
    else
        return base * power(base, exponent - 1);
}

int main()
{
    long b;
    long e;

    printf("Enter a base and an exponent ");
    scanf("%ld %ld", &b, &e);

    printf ("%ld^%ld = %ld\n", b, e, power(b, e));
}
```

```
Enter a base and an exponent 2 3
2^3 = 8
```

1st pass
base = 2
exponent = 3
return 2 * ~~power(2, 2)~~ 4

2nd pass
base = 2
exponent = 2
return 2 * ~~power(2, 1)~~ 2

3rd pass
base = 2
exponent = 1
return 2

```c
#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);
}
```

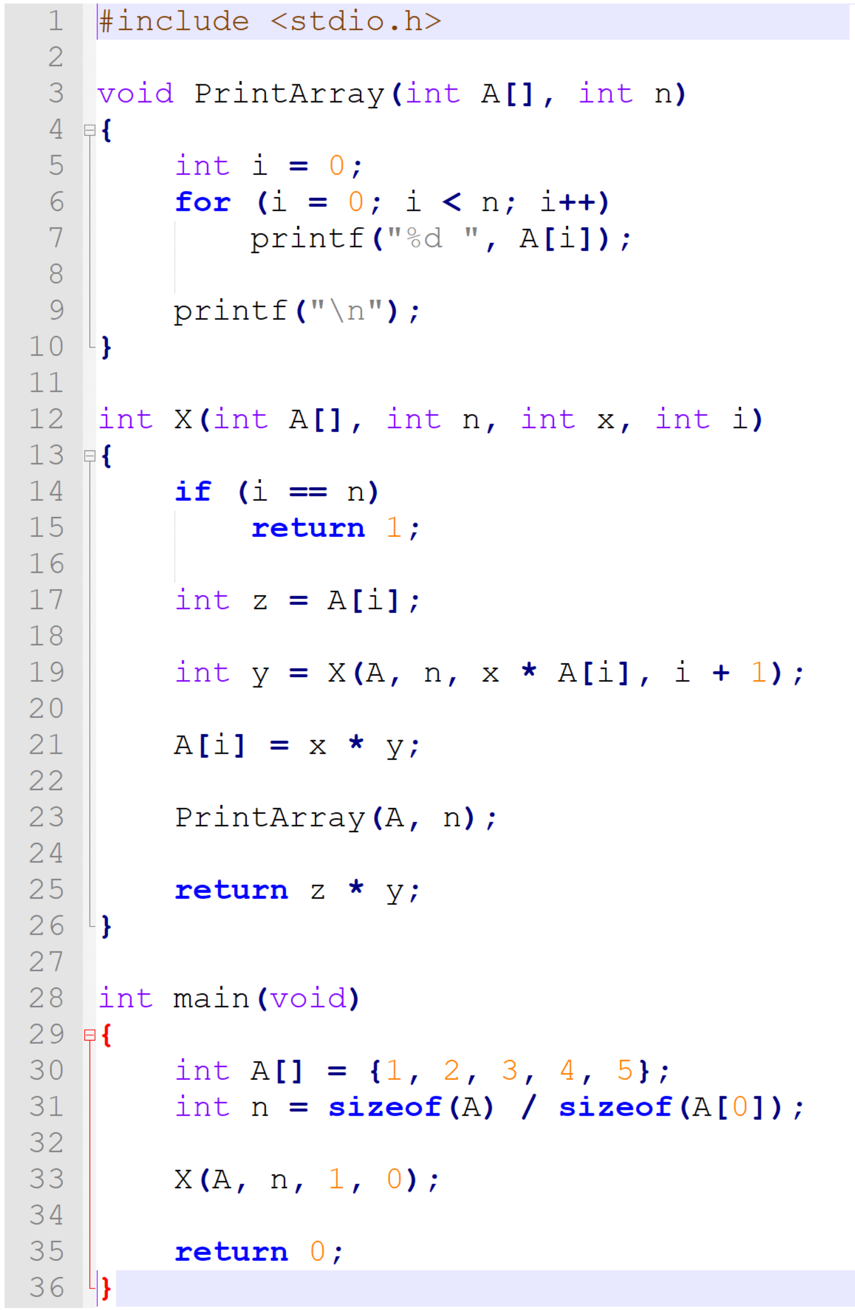
```
Pass 1
Hello(3,4,2)
if (2 < 1 && 3 > 8)
print x,y,z
print z,y,z

Pass 2
Hello(5,1,1)
if (1 < 1 && 5 > 8)
print x,y,z
print z,y,z

Pass 3
Hello(7,0,0)
if (0 < 1 && 7 > 8)
print x,y,z
print z,y,z

Pass 4
Hello(9,0,-1)
if (-1 < 1 && 9 > 8)
```

```
342
511
700
007
115
243
```

```c
#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);
}
```

```
Pass 1
Hello(2,4,2)
if (2 < 1 && 2 > 8)
print x,y,z
print z,y,z

Pass 2
Hello(4,1,1)
if (1 < 1 && 4 > 8)
print x,y,z
print z,y,z

Pass 3
Hello(6,0,0)
if (0 < 1 && 6 > 8)
print x,y,z
print z,y,z

Pass 4
Hello(8,0,-1)
if (-1 < 1 && 8 > 8)
print x,y,z
print z,y,z

Pass 5
Hello(10,0,-2)
if (-2 < 1 && 10 > 8)
```

```
242
411
600
80-1
-108
006
114
242
```



```c
#include <stdio.h>

void PrintArray(int A[], int n)
{
    int i = 0;
    for (i = 0; i < n; i++)
        printf("%d ", A[i]);

    printf("\n");
}

int X(int A[], int n, int x, int i)
{
    if (i == n)
        return 1;

    int z = A[i];

    int y = X(A, n, x * A[i], i + 1);

    A[i] = x * y;

    PrintArray(A, n);

    return z * y;
}

int main(void)
{
    int A[] = {1, 2, 3, 4, 5};
    int n = sizeof(A) / sizeof(A[0]);

    X(A, n, 1, 0);

    return 0;
}
```

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

1st 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

2nd 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

3rd 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

4th 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

5th 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

6th call
n = 5, x = 120, i = 5
i != n (5 == 5)
return 1

3rd call

n = 5, x = 2, i = 2

i != n (2 != 5)

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

y = X(A, 5, 6, 3)	y = 20
A[i] = A[2] = x * y	A[2] = x * y = 2 * 20
PrintArray	{1, 2, 40, 30, 24}
return z*y	return z*y (3*20) => 60

4th call

n = 5, x = 6, i = 3

i != n (3 != 5)

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

y = X(A, 5, 24, 4)	y = 5
A[i] = A[3] = x * y	A[3] = x * y = 6 * 5
PrintArray	{1, 2, 3, 30, 24}
return z*y	return z*y (4*5) => 20

5th call

n = 5, x = 24, i = 4

i != n (4 != 5)

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

y = X(A, 5, 120, 5)	y = 1
A[i] = A[4] = x * y	A[4] = x * y = 24 * 1
PrintArray	{1, 2, 3, 4, 24}
return z*y	return z*y (5*1) => 5

6th call

n = 5, x = 120, i = 5

i != n (5 == 5)

return 1

1st 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

y = 120
A[0] = x * y = 1 * 120
{120, 60, 40, 30, 24}
return z*y (1*120) => 120

2nd 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

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
```

```c
#include <stdio.h>

int FunctionR(int z)
{
    static int y = 0;

    if(z != 0)
    {
        y++;
        FunctionR(z/10);
    }

    return y;
}

int main(void)
{
    int x = 12335;

    printf("%d", FunctionR(x));

    return 0;
}
```

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.