CSE 2320

Week of 06/15/2020

Instructor : Donna French

Asymptotic Notation

Previously, we analyzed linear search and binary search by determining the maximum number of guesses we need to make.

The next thing we what to determine is how long these algorithms take to run.

The running times of linear search and binary search include the time needed to make and check guesses, but there's more steps to factor into how LONG the search will run.

Time matters - not just guesses.

Asymptotic Notation

The running time of an algorithm depends on how long it takes a computer to run the lines of code of the algorithm.

This running time is dependent on several factors including

- speed of the computer
- programming language
- compiler that translates the program from the programming language into code that runs directly on the computer

Asymptotic Notation

Two of the factors that determine the running time of an algorithm are

how long the algorithm takes, in terms of the size of its input

how fast a function grows with the input size

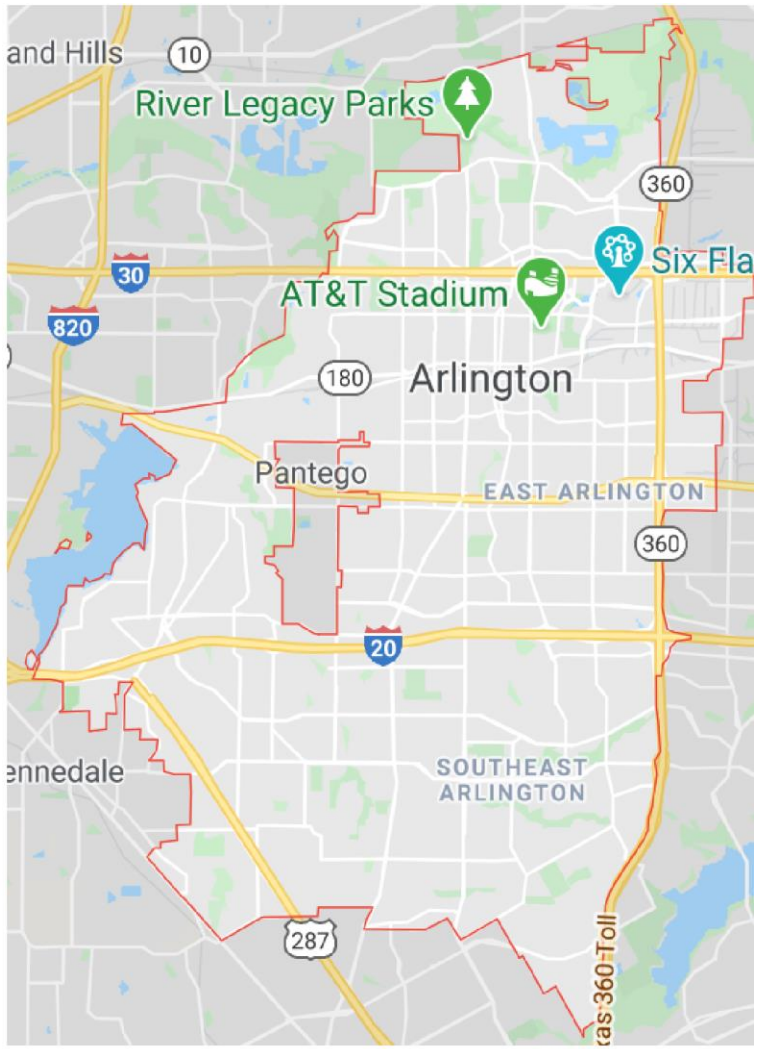
Asymptotic Notation

Let's take the first factor

how long the algorithm takes, in terms of the size of its input

We saw how the maximum number of guesses in linear search and binary search increased as the length of the array increased.

How about a GPS algorithm?



Manhattan, NY – 22.82 square miles – 12,000+ intersections

Arlington, TX – 99.69 square miles – not 12,000 intersections

Asymptotic Notation

The second factor that determines the running time of an algorithm is

how fast a function grows with the input size

This is called rate of growth of the running time.

We will discover that simplifying our functions allows us to better compare different run times and has no impact on the rate of growth.

We will see how to make a function more manageable by discarding less important parts of a function.

Asymptotic Notation

Let's compare the following 3 equations

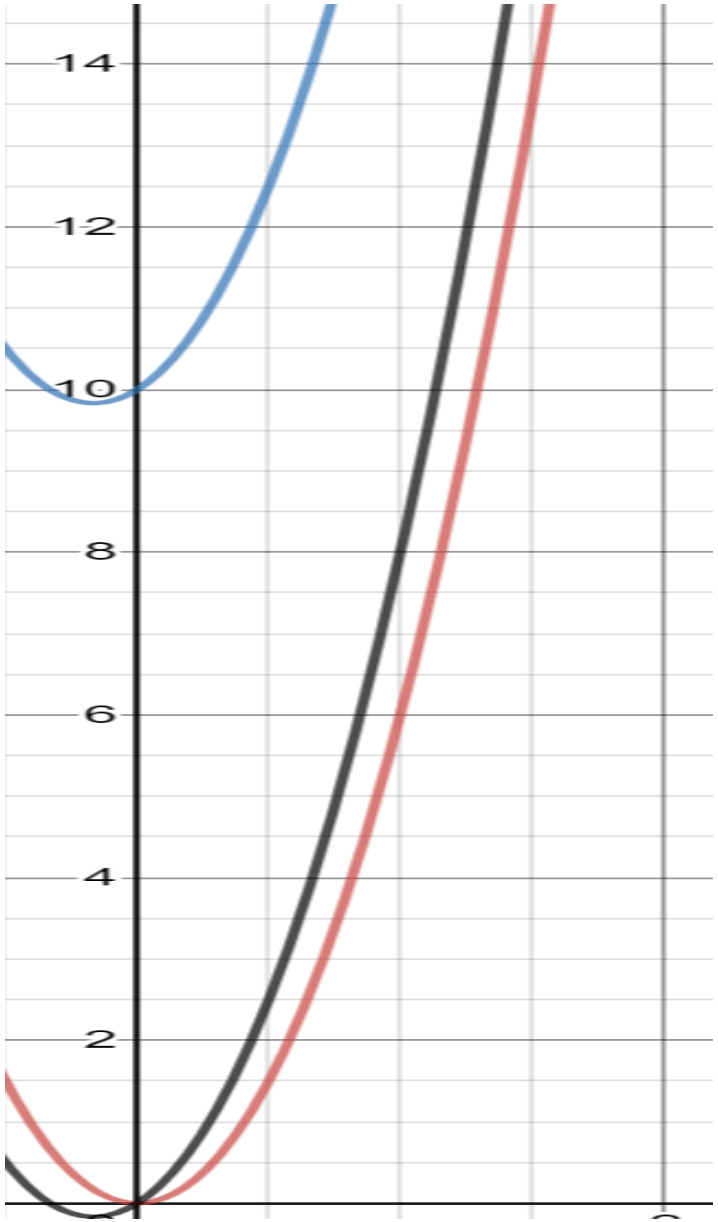
$6x^2$

$6x^2 + 2x$

$6x^2 + 2x + 10$

At smaller values, they are different..

But what happens as x grows?



Asymptotic Notation

But what happens as x grows?

$6x^2$

$6x^2 + 2x$

$\mathbf{6x^2 + 2x + 10}$

$x = 5$

$6(5)^2 = 150$

$6(5)^2 + 2(5) = 160$

$6(5)^2 + 2(5) + 10 = 170$

$x = 500$

$6(500)^2 = 1{,}500{,}000$

$6(500)^2 + 2(500) = 1{,}501{,}000$

$6(500)^2 + 2(500) + 10 = 1{,}501{,}010$

$x = 1{,}000{,}000$

$6(1{,}000{,}000)^2 = 6{,}000{,}000{,}000{,}000$

$6(1{,}000{,}000)^2 + 2(1{,}000{,}000) = 6{,}000{,}002{,}000{,}000$

$6(1{,}000{,}000)^2 + 2(1{,}000{,}000) + 10 = 6{,}000{,}002{,}000{,}010$

Asymptotic Notation

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 $c_1 n$ 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*

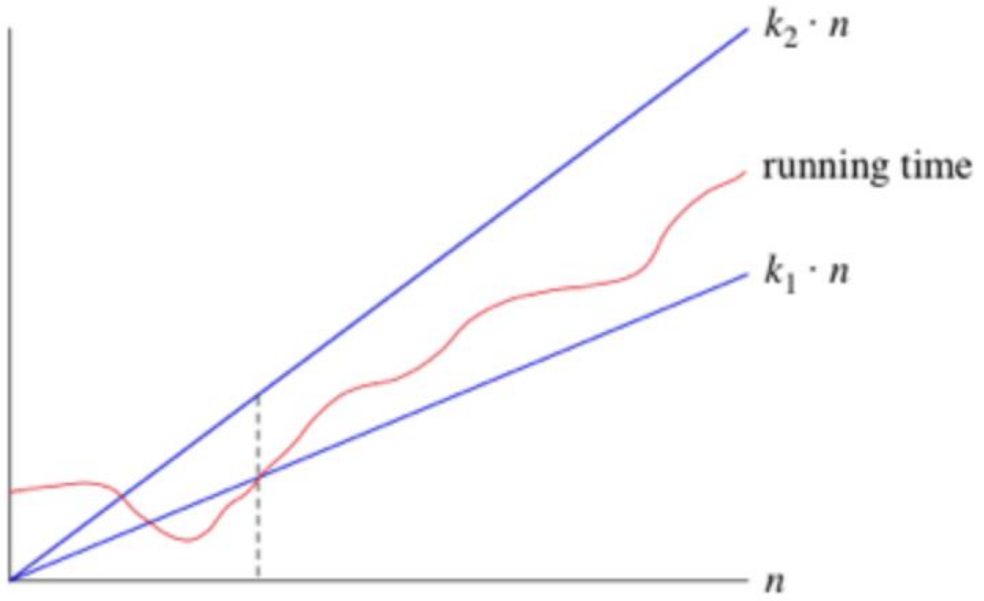
Asymptotic Notation

$\Theta(n)$

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.

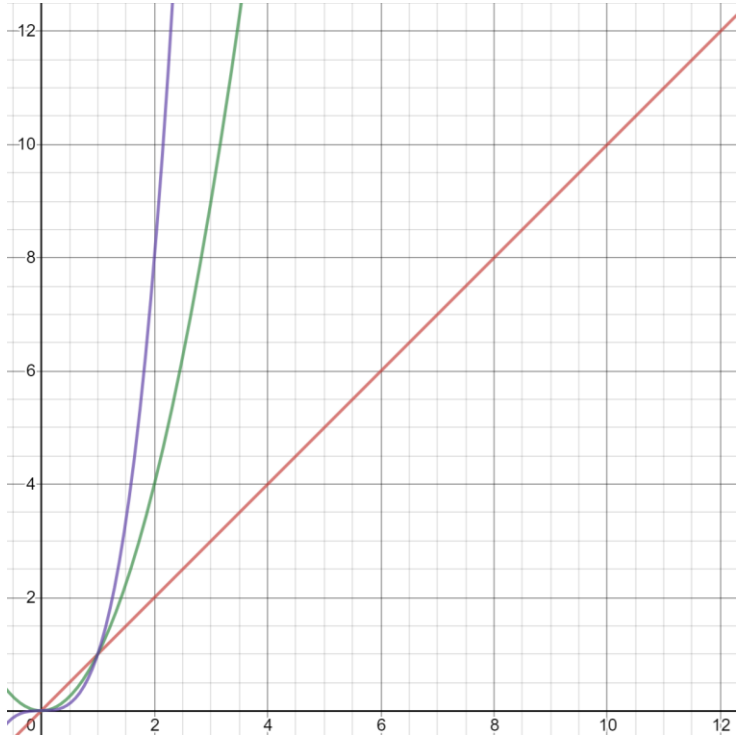
Because $n^0 = 1$, you will see this written as

$\Theta(1)$

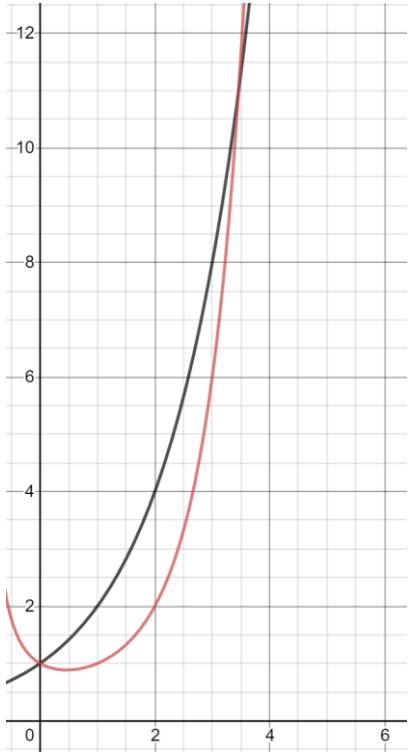
Asymptotic Notation

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

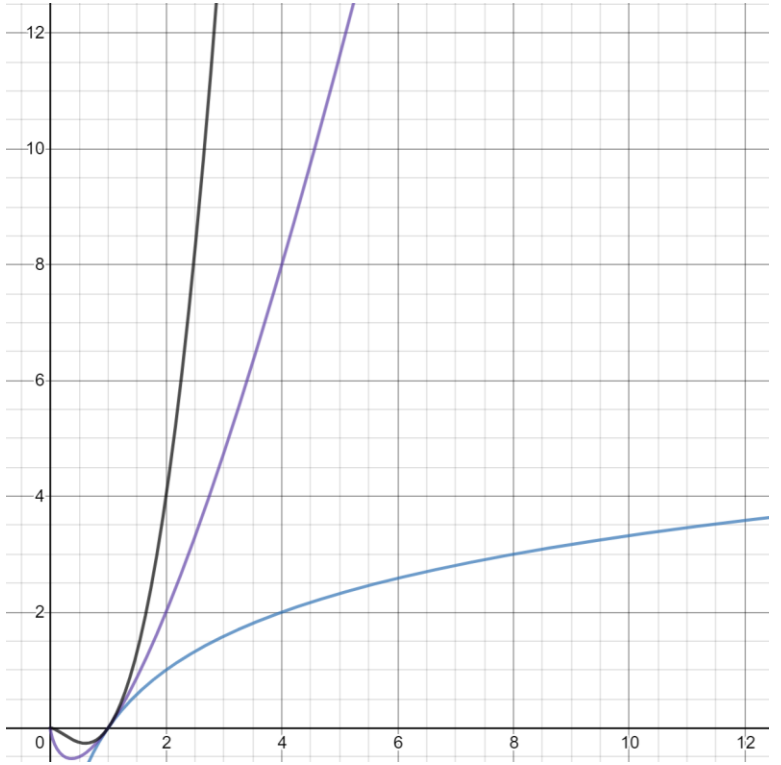
$\Theta(1)$
$\Theta(\log_2 n)$
$\Theta(n)$
$\Theta(n\log_2 n)$
$\Theta(n^2)$
$\Theta(n^2\log_2 n)$
$\Theta(n^3)$
$\Theta(2^n)$
$\Theta(n!)$



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 $c_1 n^2$

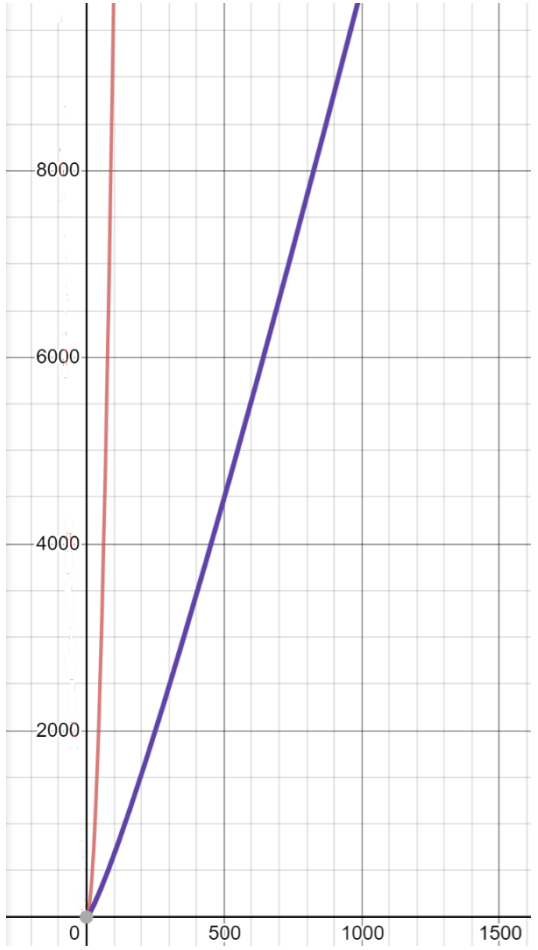
Merge Sort $c_2 n \log_2 n$

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 $c_1 n^2$

$c_1 < c_2$

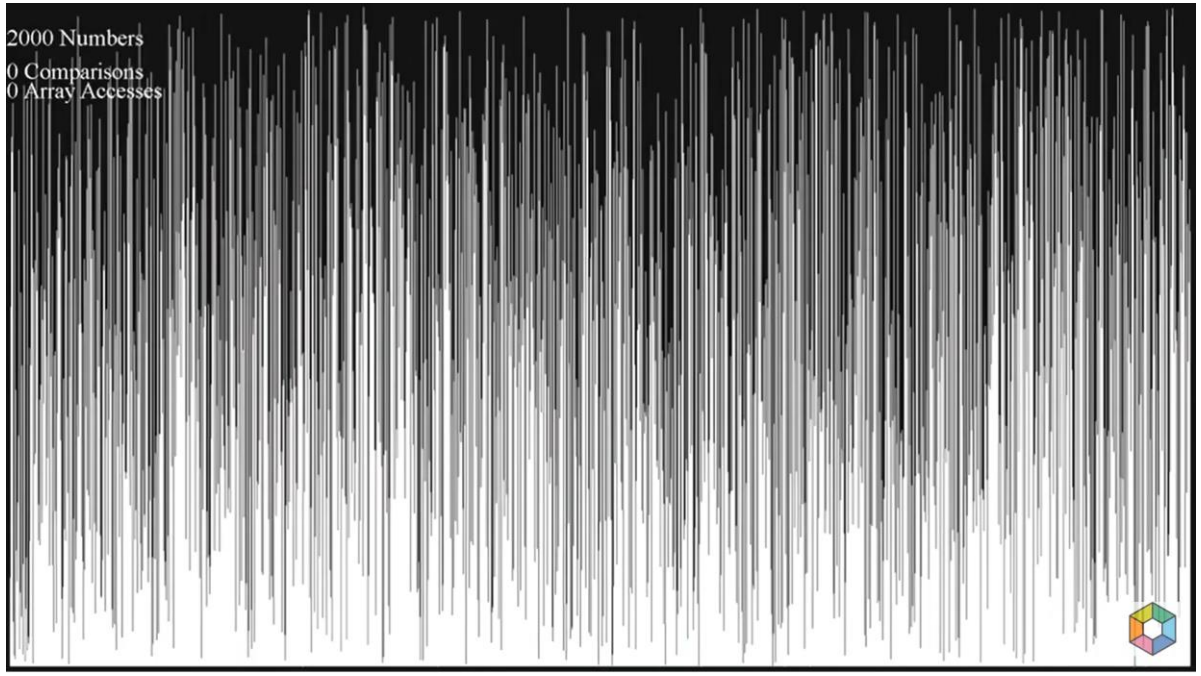
Merge Sort $c_2 n \log_2 n$

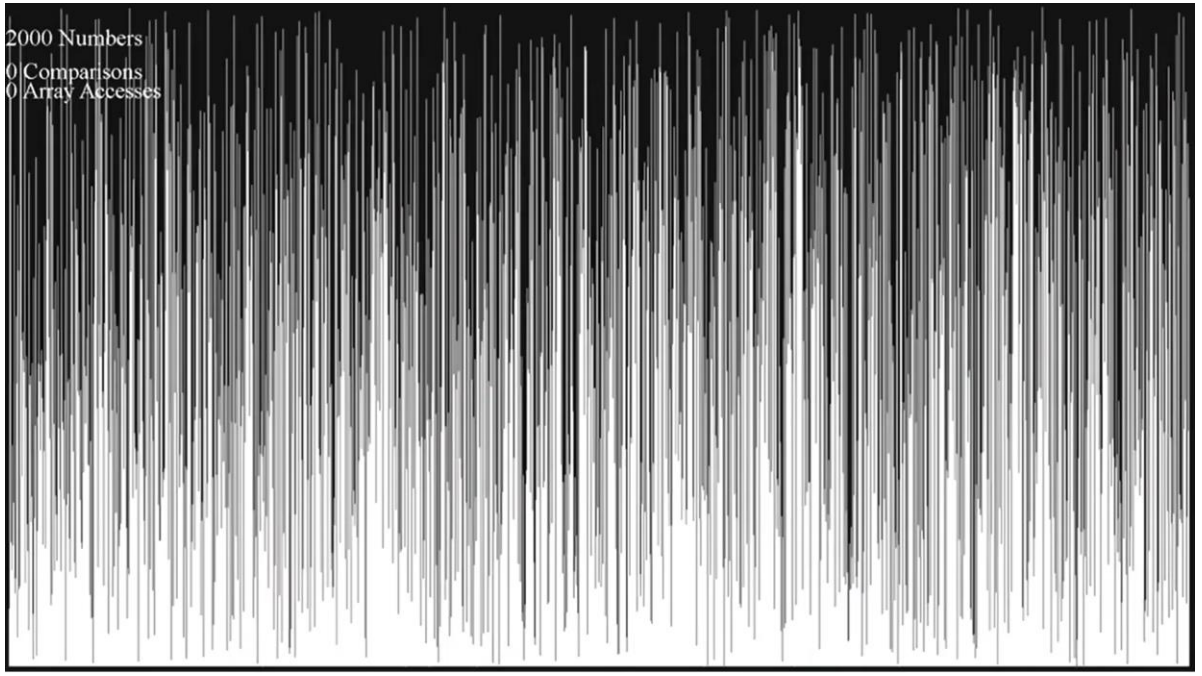
If n is 1,000,000, then n^2 is 1,000,000,000,000 and $n\log_2 n$ is ~200

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_1n^2 n = 2000

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

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

$0.72n^2$

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
$c_2 \approx 2.89$

$2.89n \log_2 n$

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.

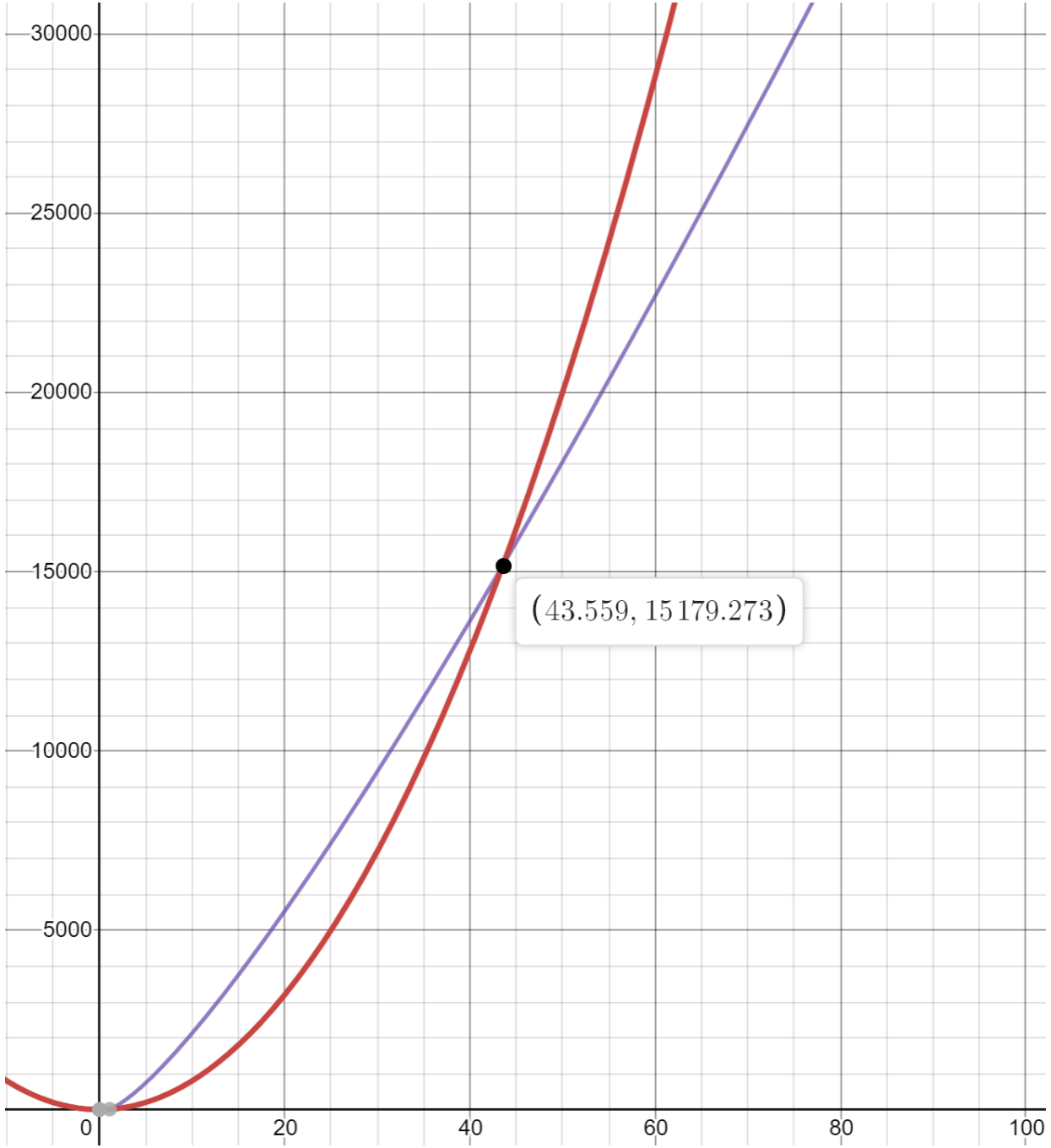
For which values of n does insertion sort beat merge sort?

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	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

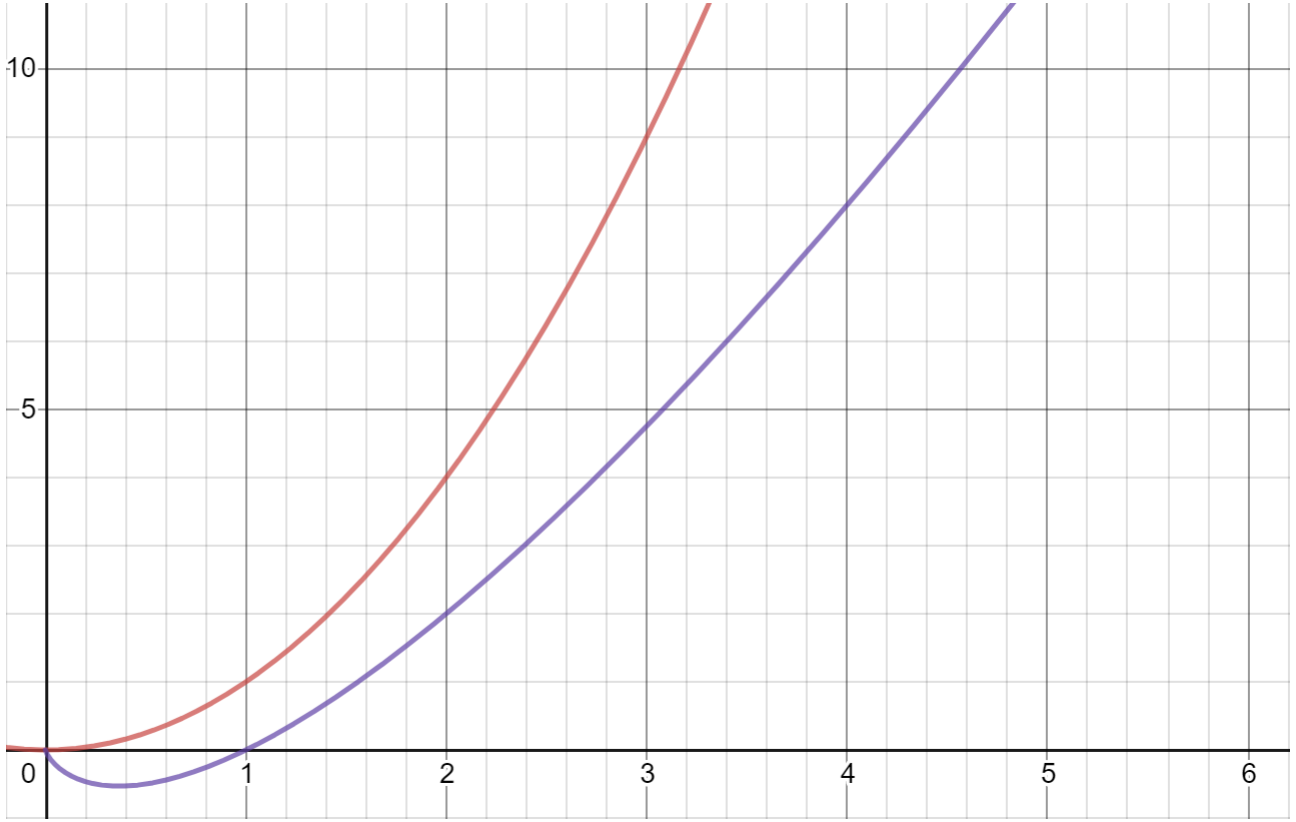
The Role of Algorithms in Computing

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

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

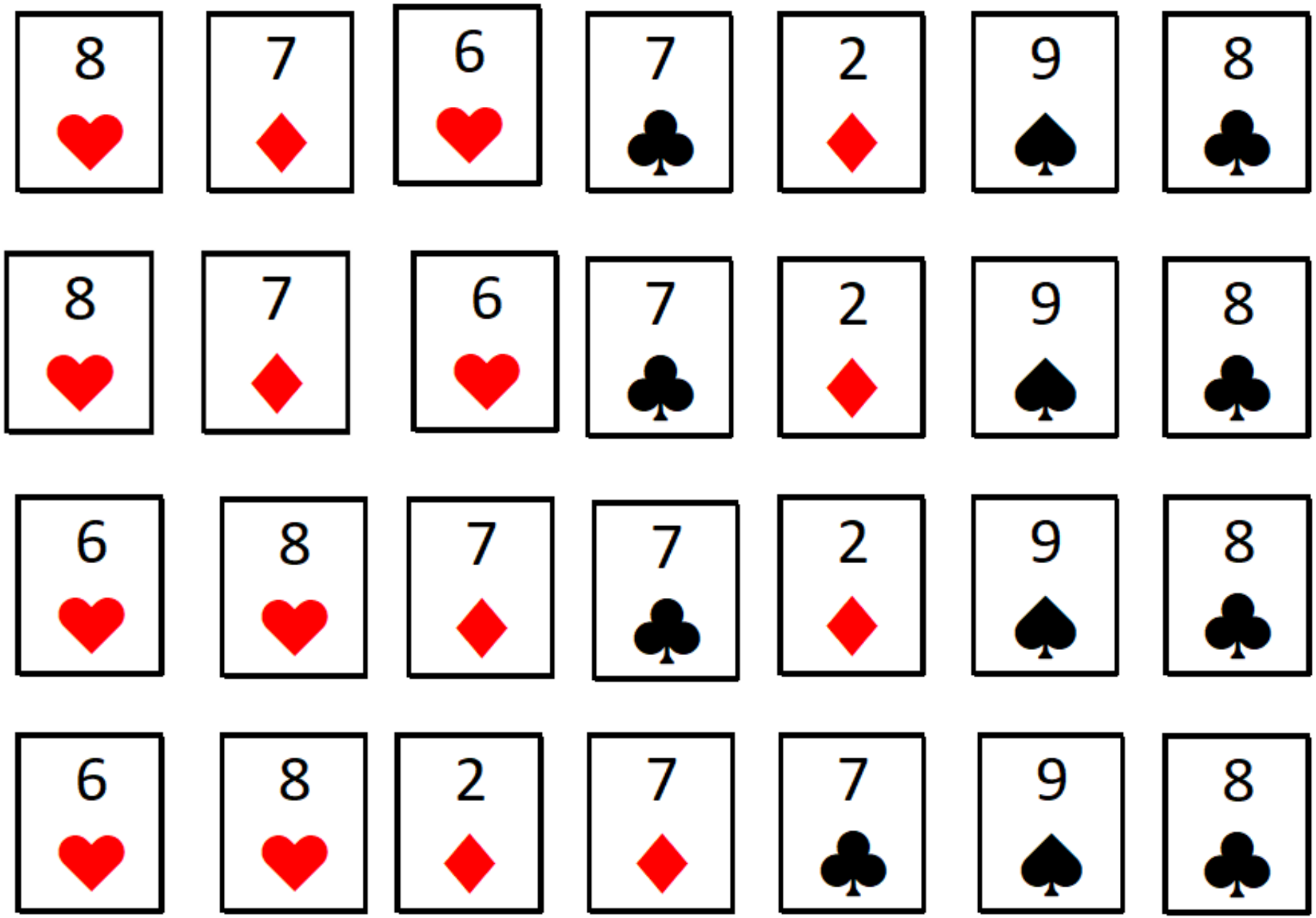
Insertion Sort algorithm sorts the input numbers in place.

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

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

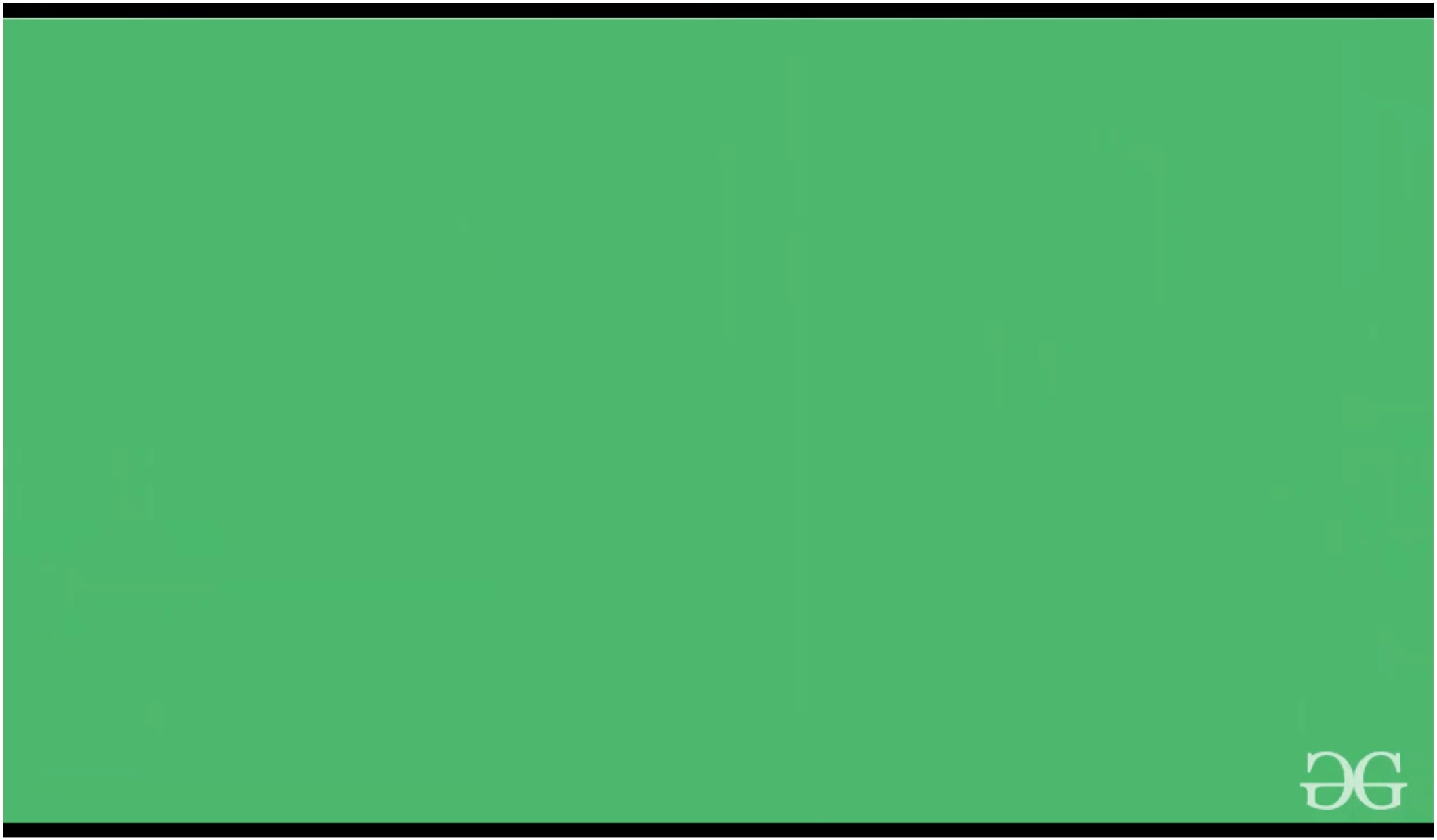
i = 3 5 will move to the beginning and all other elements from 11 to 13 will move one position ahead of their current position.

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

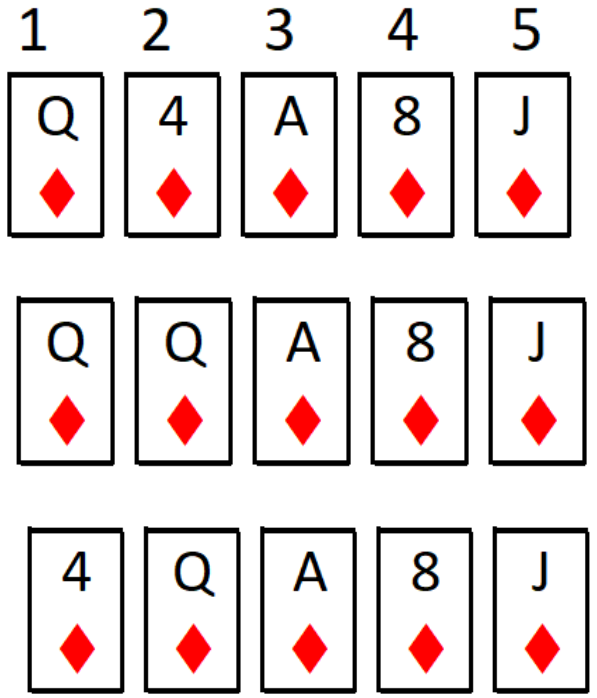


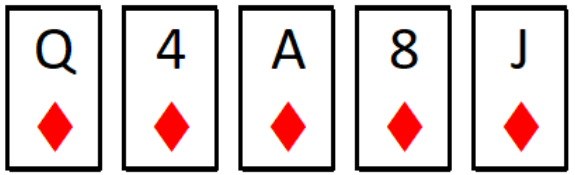
Insertion Sort

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]



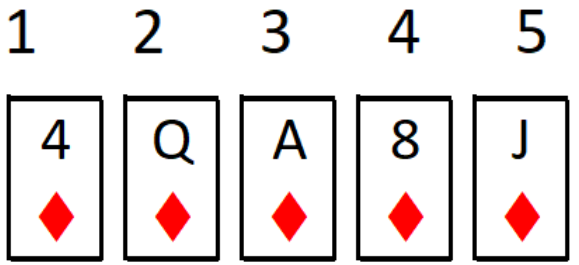


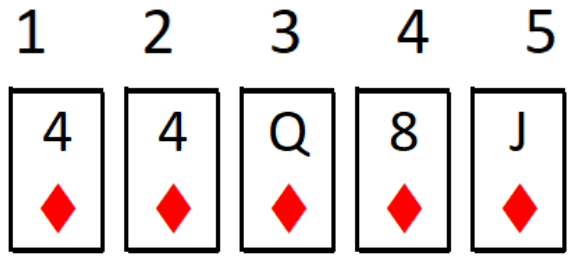
Insertion Sort

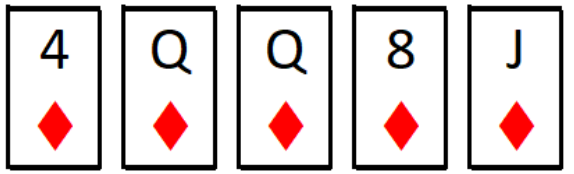
n is the number of elements to sort

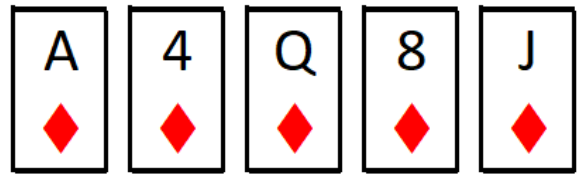
```
for j = 2 to n

    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]









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

$A[j] \leq key$

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	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

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

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

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

```
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

Analysis of Insertion Sort

t_j is the number of times the while loop runs

Worst Case Scenario

Running time

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

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

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

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 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

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

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

Analysis of Insertion Sort

t_j is the number of times the while loop runs

Worst Case Scenario

Running time

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

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

Analysis of Insertion Sort

t_j is the number of times the while loop runs

Worst Case Scenario

Running time

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

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

Analysis of Insertion Sort

t_j is the number of times the while loop runs

Worst Case Scenario

Running time

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

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

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

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 $A[1..j\text{-}1]$ and it's greater than the other half.

On average, the while loop has to look halfway through the sorted subarray $A[1..j\text{-}1]$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 Θ-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-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.

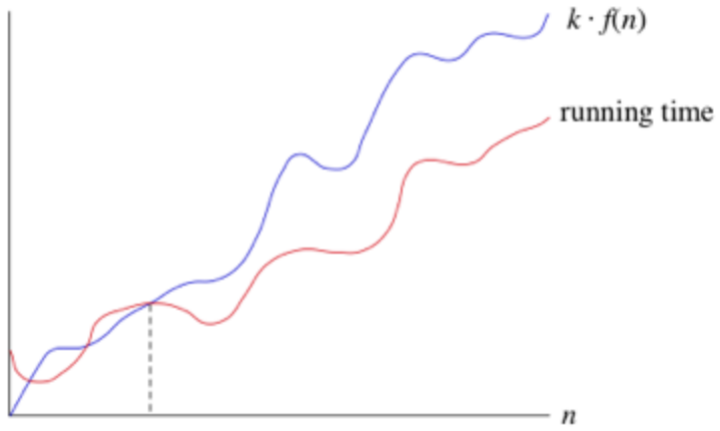
The running time of binary search is never worse than $\Theta(\log_2 n)$, but it's sometimes better.

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

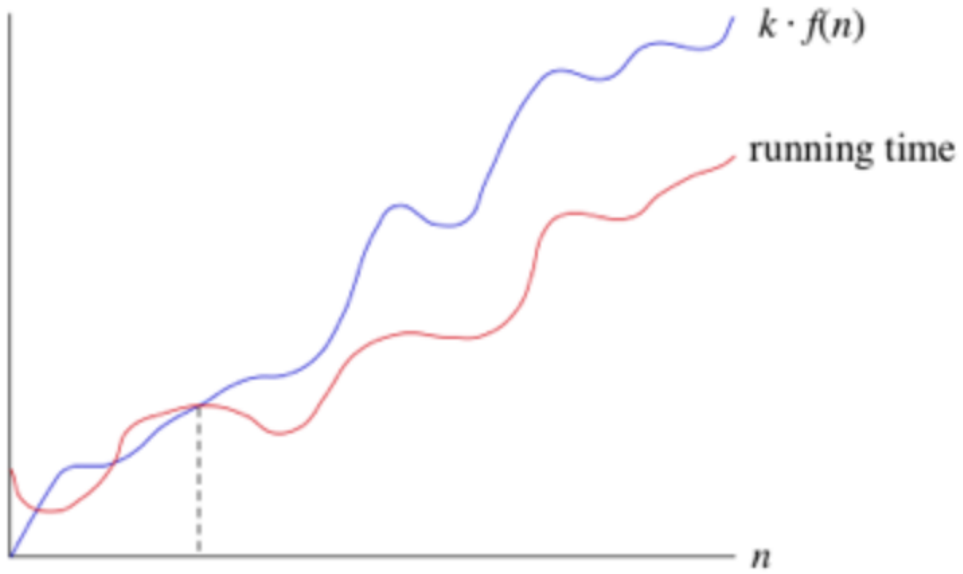
Asymptotic Notation

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

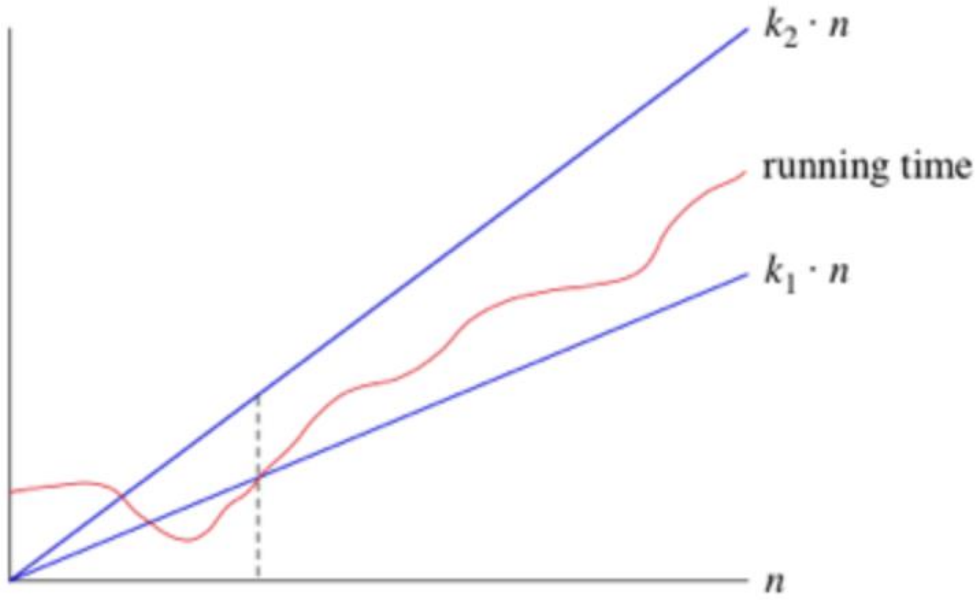
We use big-O notation for **asymptotic upper bounds**, since it bounds the growth of the running time from above for large enough input sizes.



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

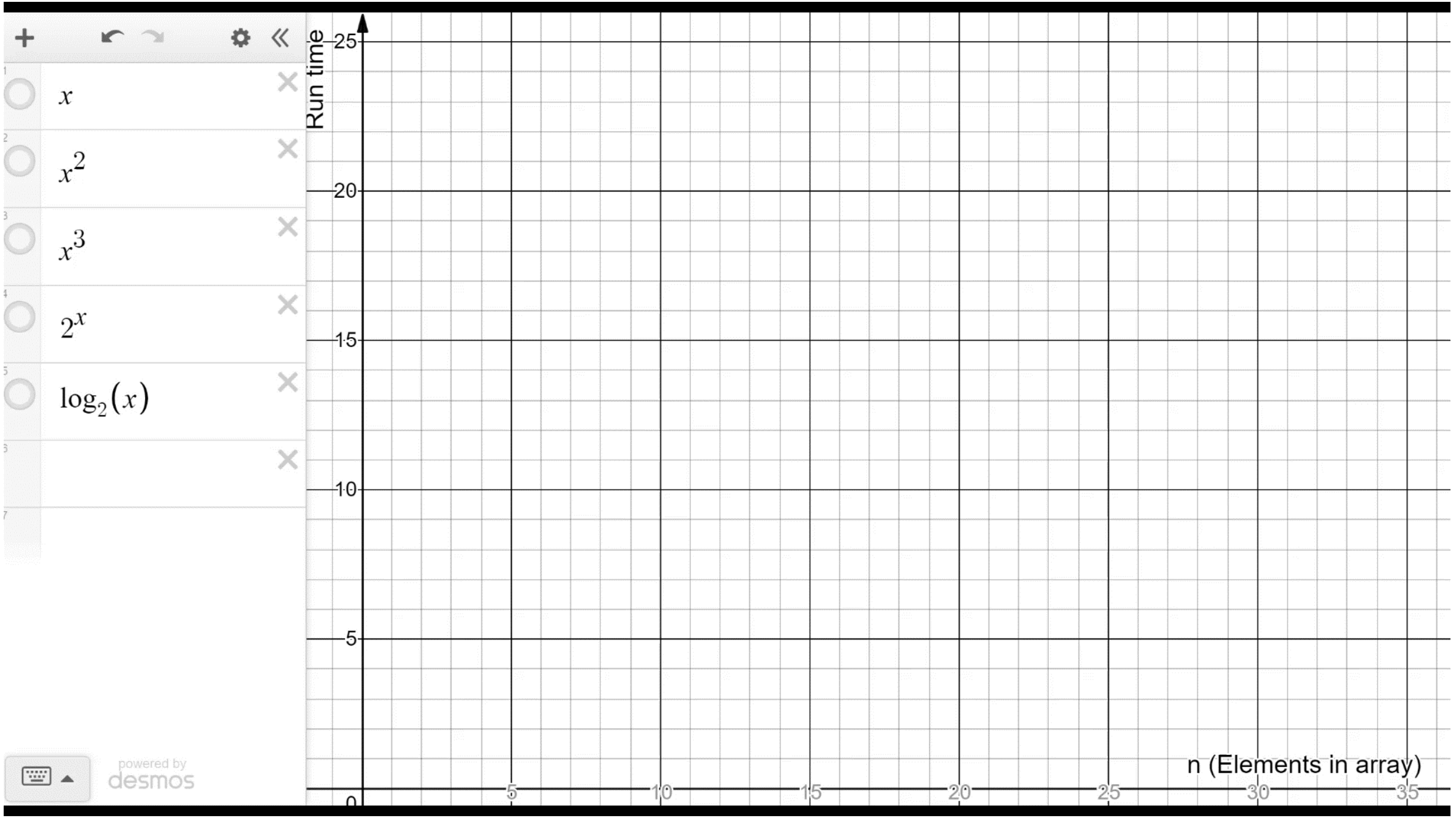
The worst-case running time of binary search is $\Theta(\log_2 n)$ time but it would also be accurate (not precise) to state that binary search has a run time of $O(n)$, $O(n^2)$, $O(n^3)$, and $O(2^n)$.

For example, if n = 4…

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

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.


x
x^2
x^3
2^x
$\log_2(x)$
powered by desmos
Run time
25
20
15
10
5
0
5
10
15
20
25
30
35
n (Elements in array)

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.

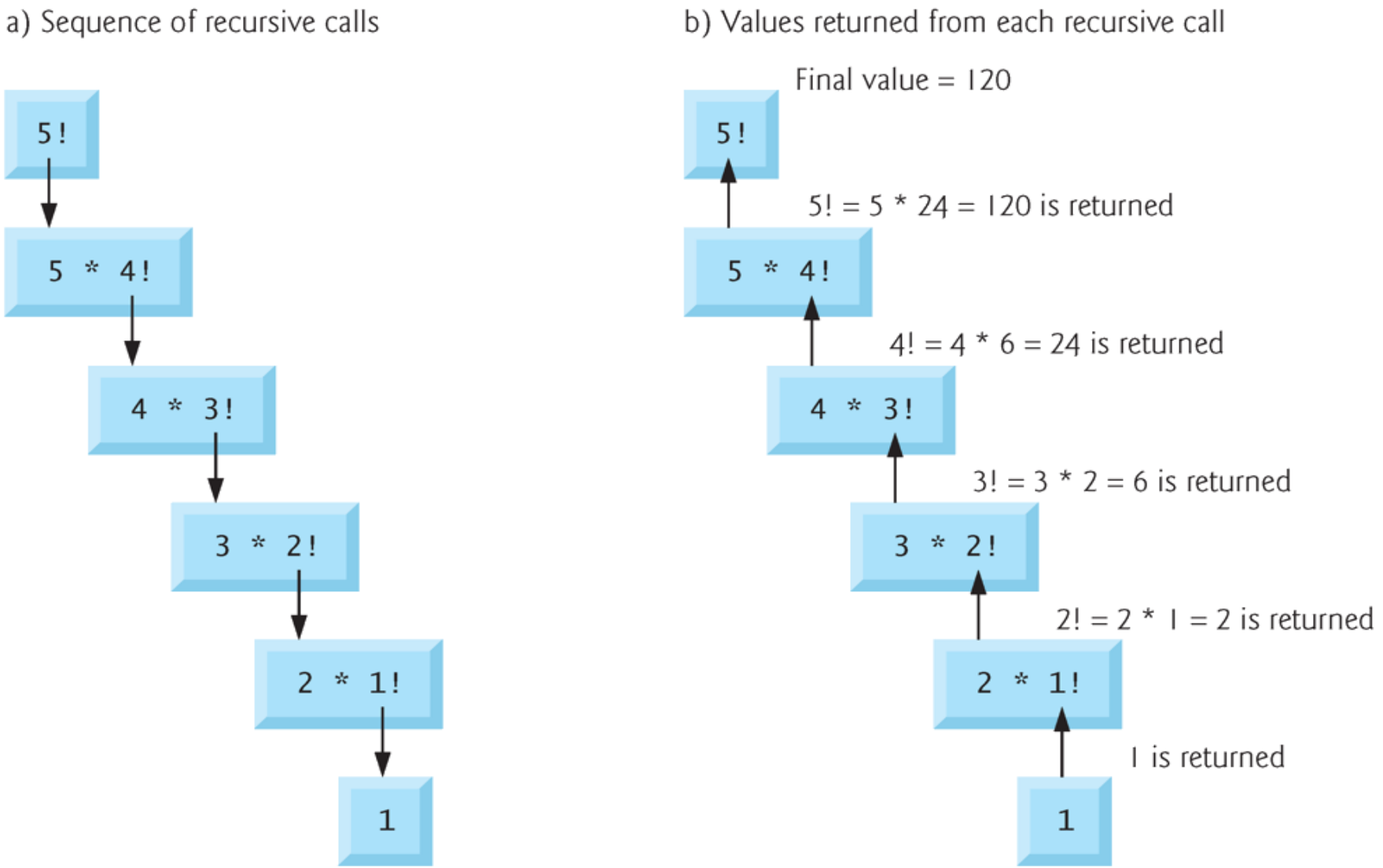
A simple example of a mathematical recursion is factorial

```
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));
}
```



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.

```
(gdb) bt
#0  factorial (n=0) at frDemo.c:6
#1  0x00000000004004fd in factorial (n=1) at frDemo.c:9
#2  0x00000000004004fd in factorial (n=2) at frDemo.c:9
#3  0x00000000004004fd in factorial (n=3) at frDemo.c:9
#4  0x00000000004004fd in factorial (n=4) at frDemo.c:9
#5  0x000000000040053d in main () at frDemo.c:19
```

After processing `n=0`

```
#0  0x00000000004004fd in factorial (n=1) at frDemo.c:9
#1  0x00000000004004fd in factorial (n=2) at frDemo.c:9
#2  0x00000000004004fd in factorial (n=3) at frDemo.c:9
#3  0x00000000004004fd in factorial (n=4) at frDemo.c:9
#4  0x000000000040053d in main () at frDemo.c:19
```

After processing `n=1`

```
#0  0x00000000004004fd in factorial (n=2) at frDemo.c:9
#1  0x00000000004004fd in factorial (n=3) at frDemo.c:9
#2  0x00000000004004fd in factorial (n=4) at frDemo.c:9
#3  0x000000000040053d in main () at frDemo.c:19
```

After processing `n=2`

```
#0  0x00000000004004fd in factorial (n=3) at frDemo.c:9
#1  0x00000000004004fd in factorial (n=4) at frDemo.c:9
#2  0x000000000040053d in main () at frDemo.c:19
```

After processing `n=3`

```
#0  0x00000000004004fd in factorial (n=4) at frDemo.c:9
#1  0x00000000040053d in main () at frDemo.c:19
```

After processing `n=4`

```
#0  0x00000000040053d 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;
    }
}
```

```
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;
}
```

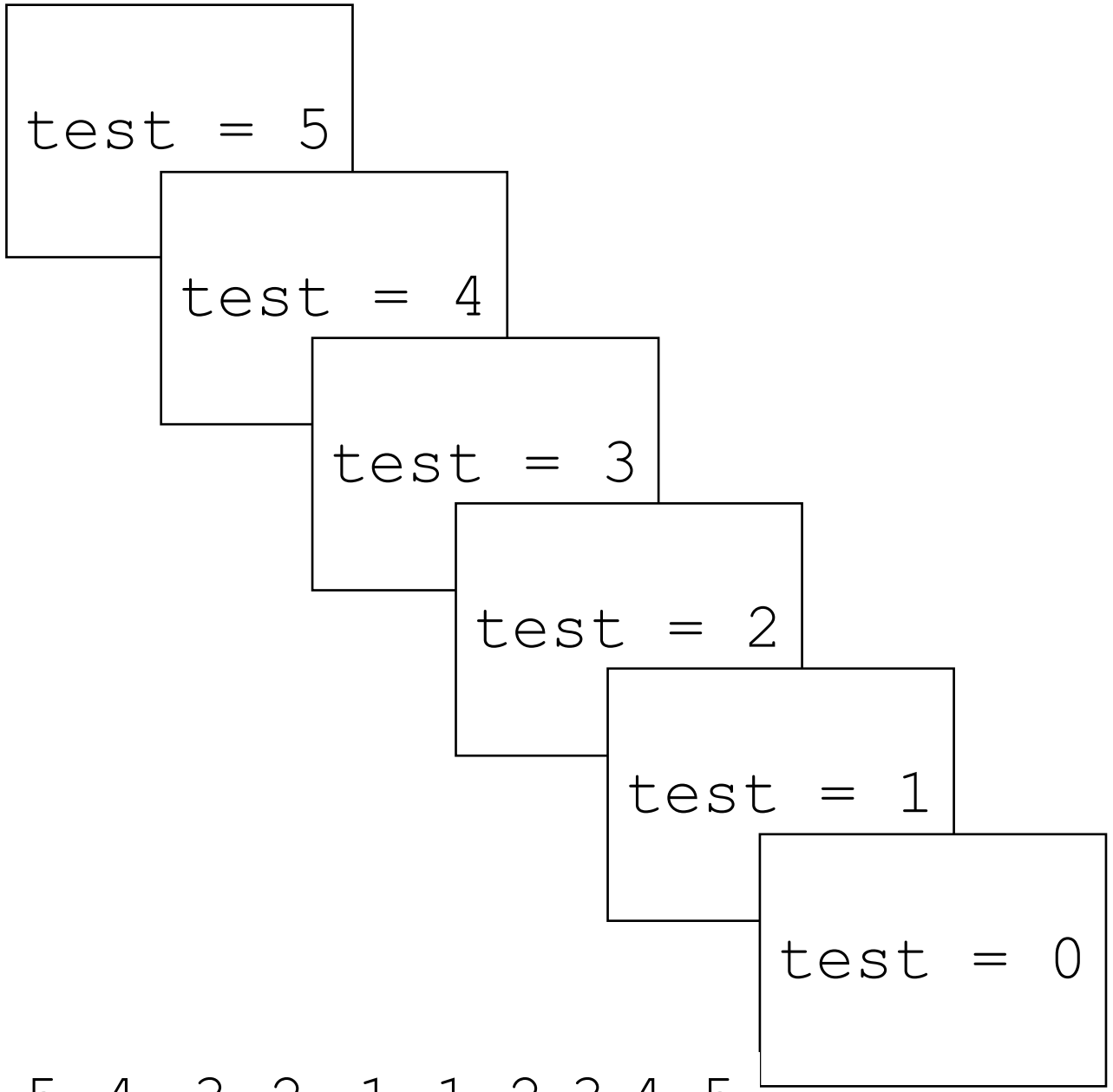

frenchdm@omega:~
[frenchdm@omega ~]$

```
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;
}
```

What is the condition that makes it stop?

```
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, …

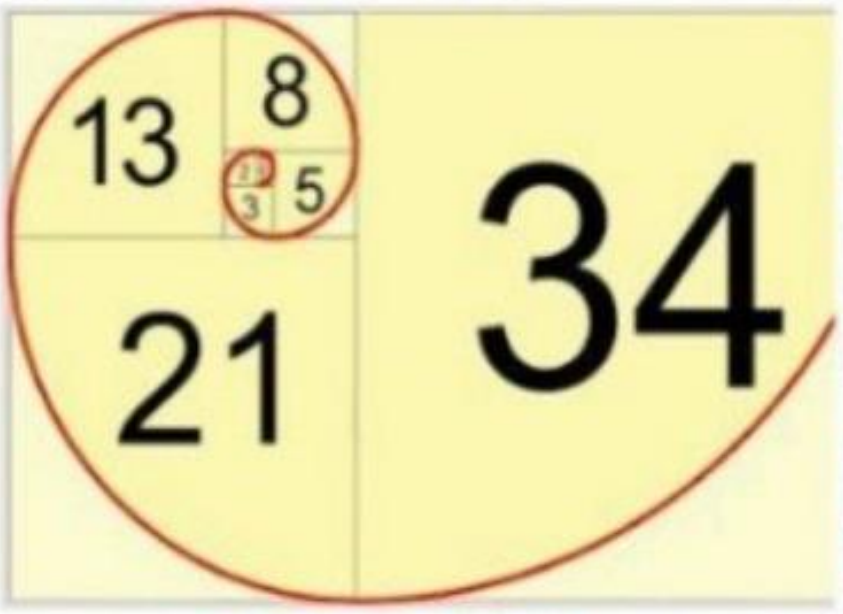
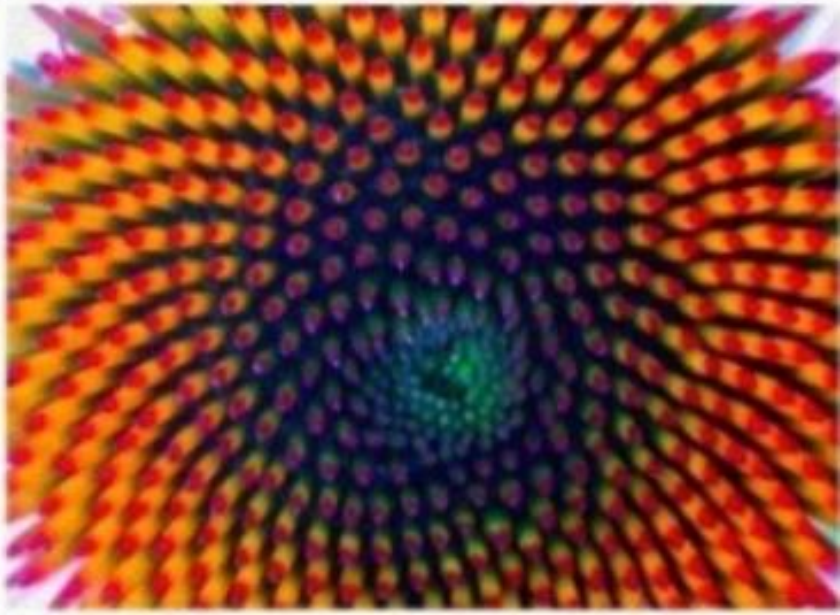
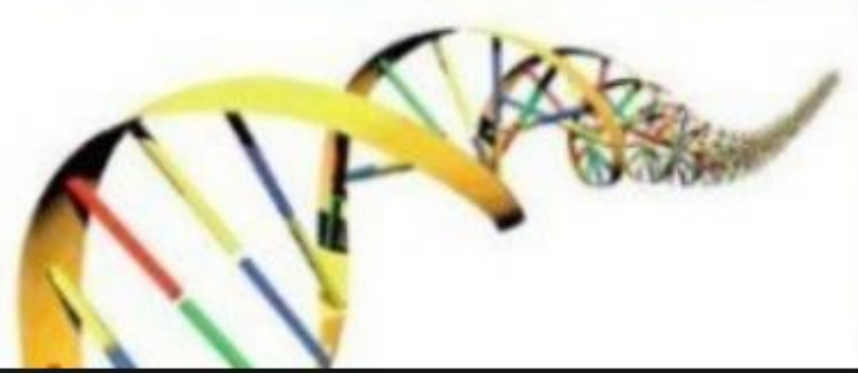
begins with 0 and 1 and has the property that each subsequent Fibonacci number is the sum of the previous two Fibonacci numbers.

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….






13
8
5
21
34



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);
     }
}
```

```
Enter an integer: 0

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

fibonacci (n=0) at recur2Demo.c:6

4       unsigned long long int fibonacci(unsigned int n)
5       {
6               if (n == 0 || n == 1)
7               {
8                       return n;
9               }


Fibonacci(0) = 0
Fibonacci(1) = 1
Fibonacci(2) = 1
Fibonacci(3) = 2
Fibonacci(4) = 3
Fibonacci(5) = 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.

Designing Algorithms

Insertion sort uses an incremental approach

> having sorted the subarray $A[1..j-1]$, we inserted the single element $A[j]$ into its proper place, yielding the sorted subarray $A[1..j]$

We are going to examine an alternative design approach called

> divide and conquer

Designing Algorithms

Many useful algorithms are recursive in structure

To solve a given problem, an algorithm calls itself recursively one or more times to deal with closely related subproblems.

These types of algorithms follow a divide and conquer approach

They break the problem into several subproblems that are similar to the original problem but smaller in size, solve the subproblems recursively and the combine these solutions to create a solution to the original problem.

The Divide and Conquer Approach

Divide the problem into a number of subproblems that are smaller instances of the same problem.

Conquer the subproblems by solving them recursively. If the subproblems are small enough, just solve them by brute force.

Combine the subproblem solutions to give a solution to the original problem

Merge Sort

Merge Sort is a sorting algorithm based on divide and conquer.

Its worst-case running time has a lower order of growth than insertion sort.

Because we are dealing with subproblems, we state each subproblem as sorting a subarray $A[p \,..\, r]$

Initially, $p = 1$ and $r = n$, but these values change as we recurse through subproblems

Merge Sort

To sort $A[p \,..\, r]$

Divide by splitting into two subarrays $A[p \,..\, q]$ and $A[q+1 \,..\, r]$ where q is the halfway point of $A[p \,..\, r]$

Conquer by recursively sorting the two subarrays $A[p \,..\, q]$ and $A[q+1 \,..\, r]$

Combine by merging the two sorted subarrays $A[p \,..\, q]$ and $A[q+1 \,..\, r]$ to produce a single sorted subarray $A[p \,..\, r]$. To accomplish this step, we'll define a procedure `Merge(A, p, q, r)`

The recursion bottoms out when the subarray has just 1 element, so that it's trivially sorted – can't divide 1 value.

Merge Sort

```
MergeSort(A, p, r)

    if p < r                          // check for base case

        q = (p + r)/2                 // divide

        MergeSort (A, p, q)           // conquer
        MergeSort(A, q+1, r)          // conquer

        Merge (A, p, q, r)            // combine
```

```
int main()
{
    int arr[] = {12, 11, 5, 13, 7, 6};
    int arr_size = sizeof(arr)/sizeof(arr[0]);

    printf("Given array is \n");
    printArray(arr, arr_size);

    mergeSort(arr, 0, arr_size - 1);

    printf("\nSorted array is \n");
    printArray(arr, arr_size);

    return 0;
}
```