QuickSort

CIS 15 : Spring 2007

Functionalia

TEA!

HW 1 is **DUE FRIDAY 23rd, 11:59 PM**

- Do the BASIC Program First!
- Turn in Basic Program and Challenges Seperately

Today:

- Binary Search Example
- QuickSort

Submitting Homework

Submit Homework by running a script on your homework file:

`$ ~chipp/Public/bin/hw1-submit hw1.cpp`

(**`hw1.cpp`** is the name of your HomeWork 1 file)

Don't forget the tilde (~) at the beginning of the command!

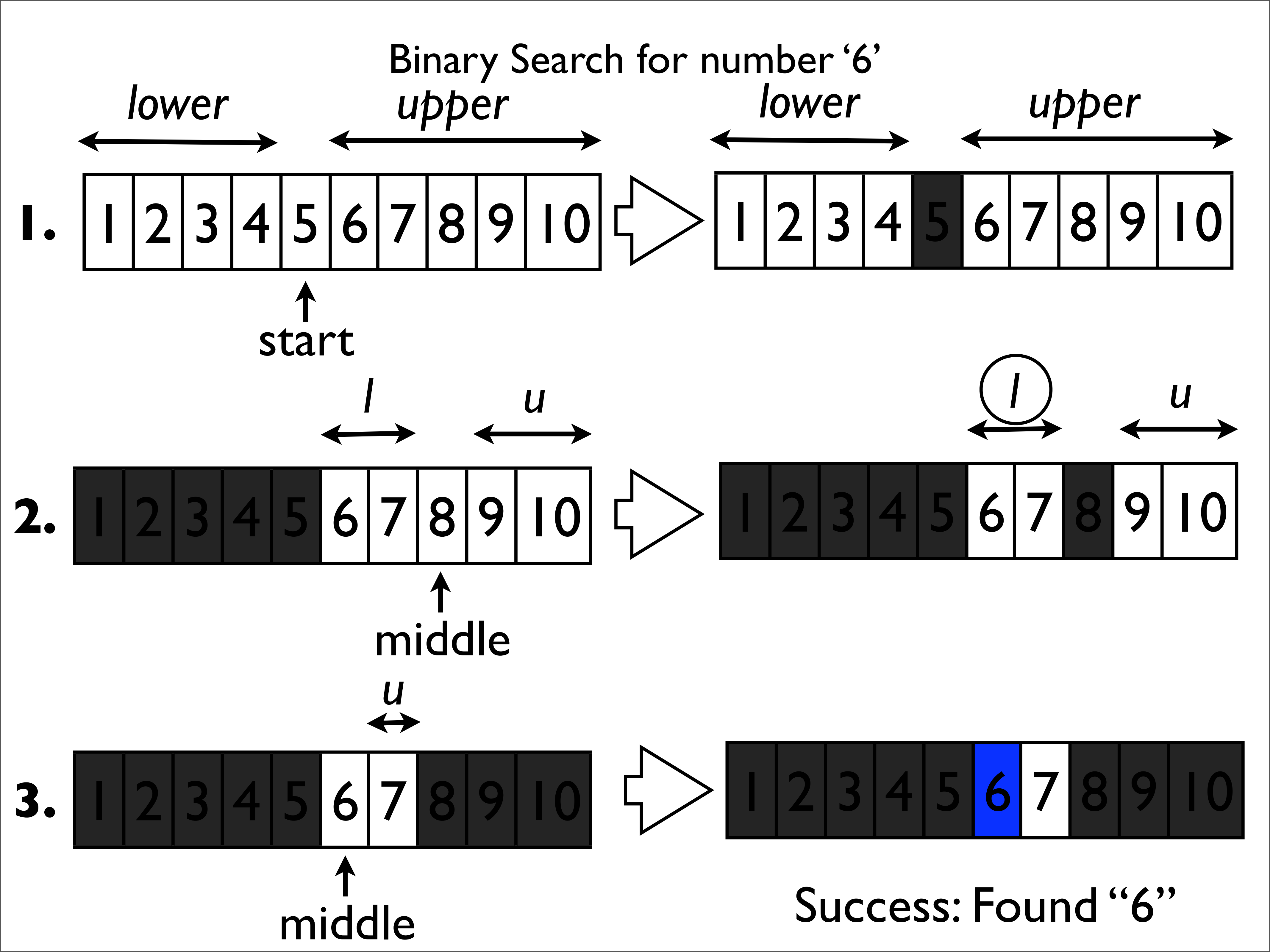
Alternatively, try this:

`$ /users1/chipp/Public/bin/hw1-submit hw1.cpp`

And if all else fails, e-mail your **`hw1.cpp`** file to me at:

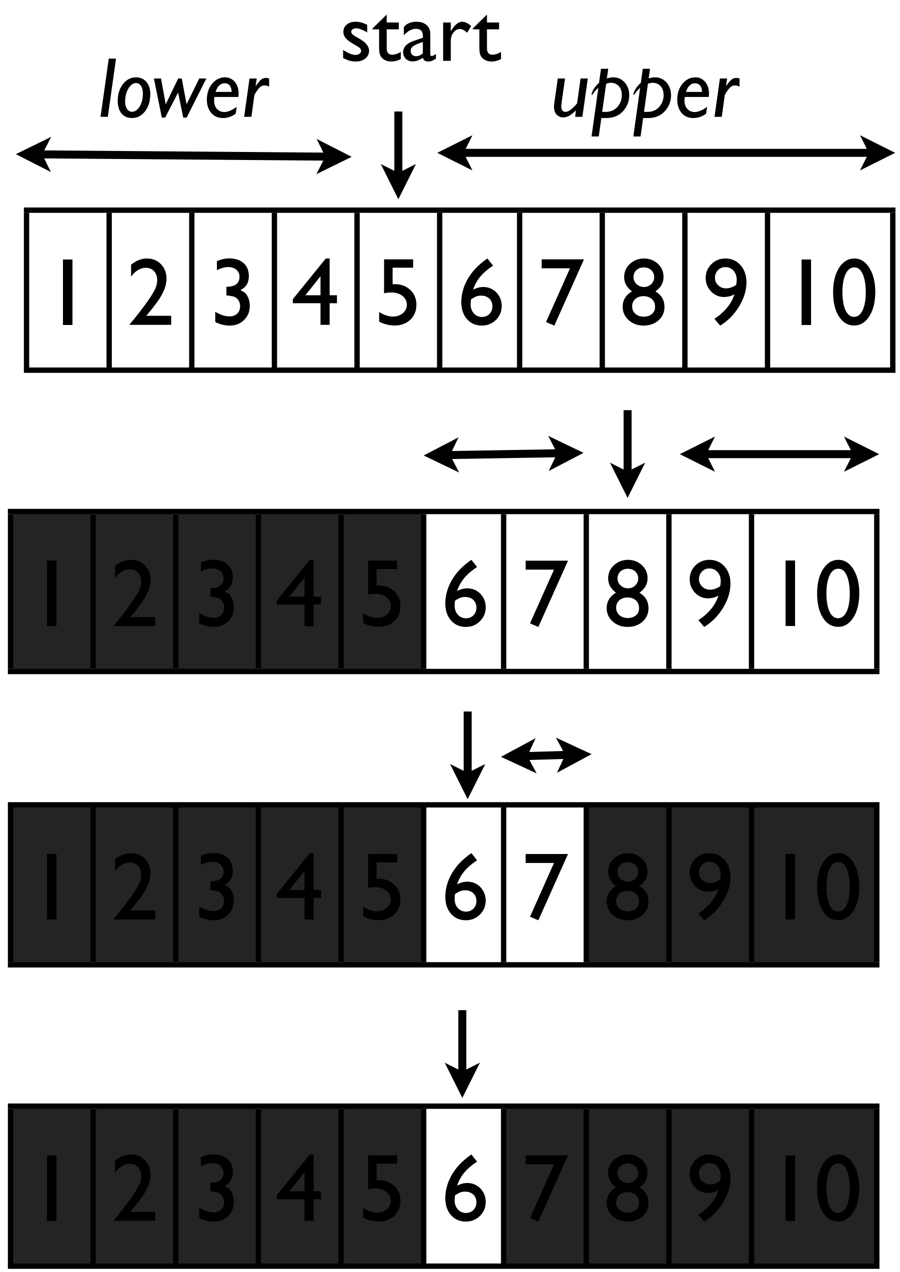
chipp@sci.brooklyn.cuny.edu

Finally, please check that your program runs on the UNIX machines!

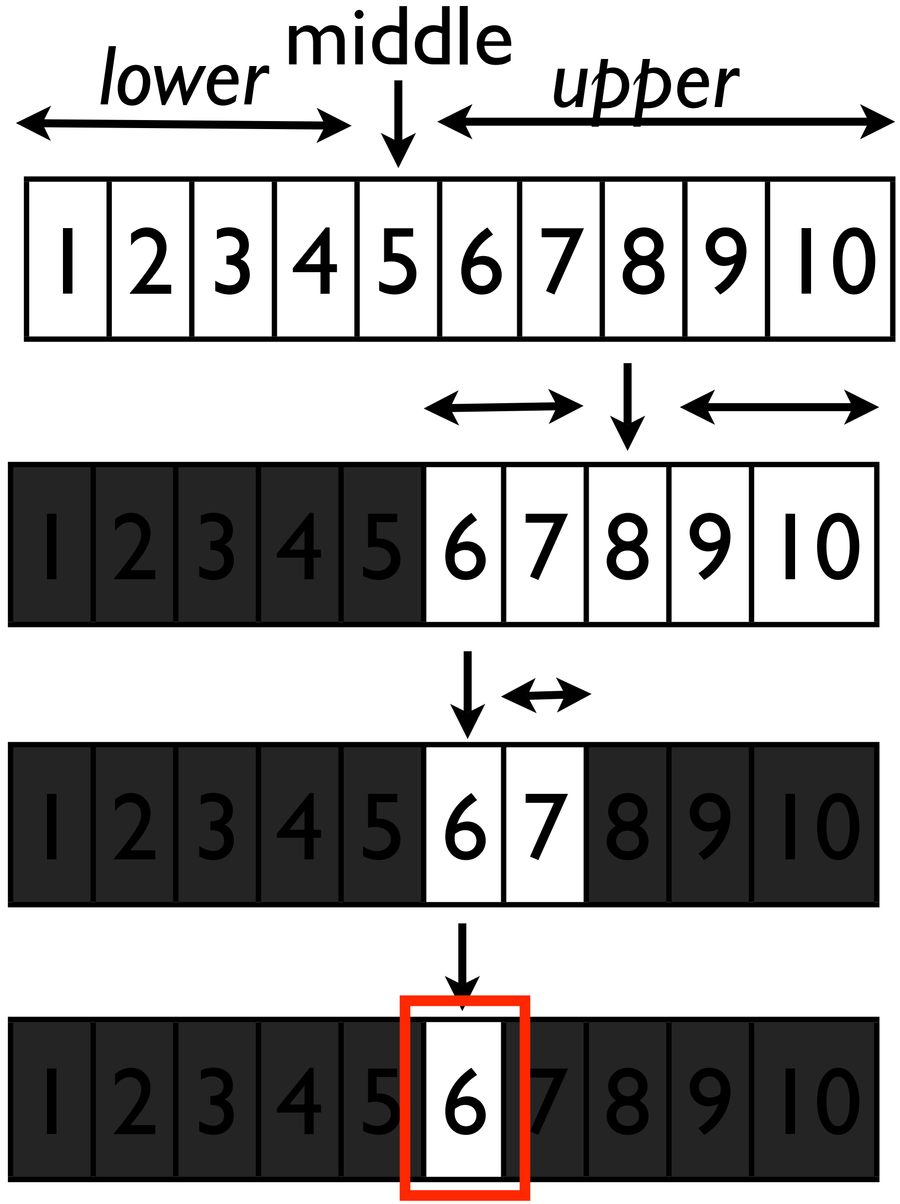


Binary Search for number '6'
lower
upper
lower
upper
1.
1 2 3 4 5 6 7 8 9 10
1 2 3 4 5 6 7 8 9 10
start
l
u
l
u
2.
1 2 3 4 5 6 7 8 9 10
1 2 3 4 5 6 7 8 9 10
middle
u
3.
1 2 3 4 5 6 7 8 9 10
1 2 3 4 5 6 7 8 9 10
middle
Success: Found "6"

What is the Base Case?

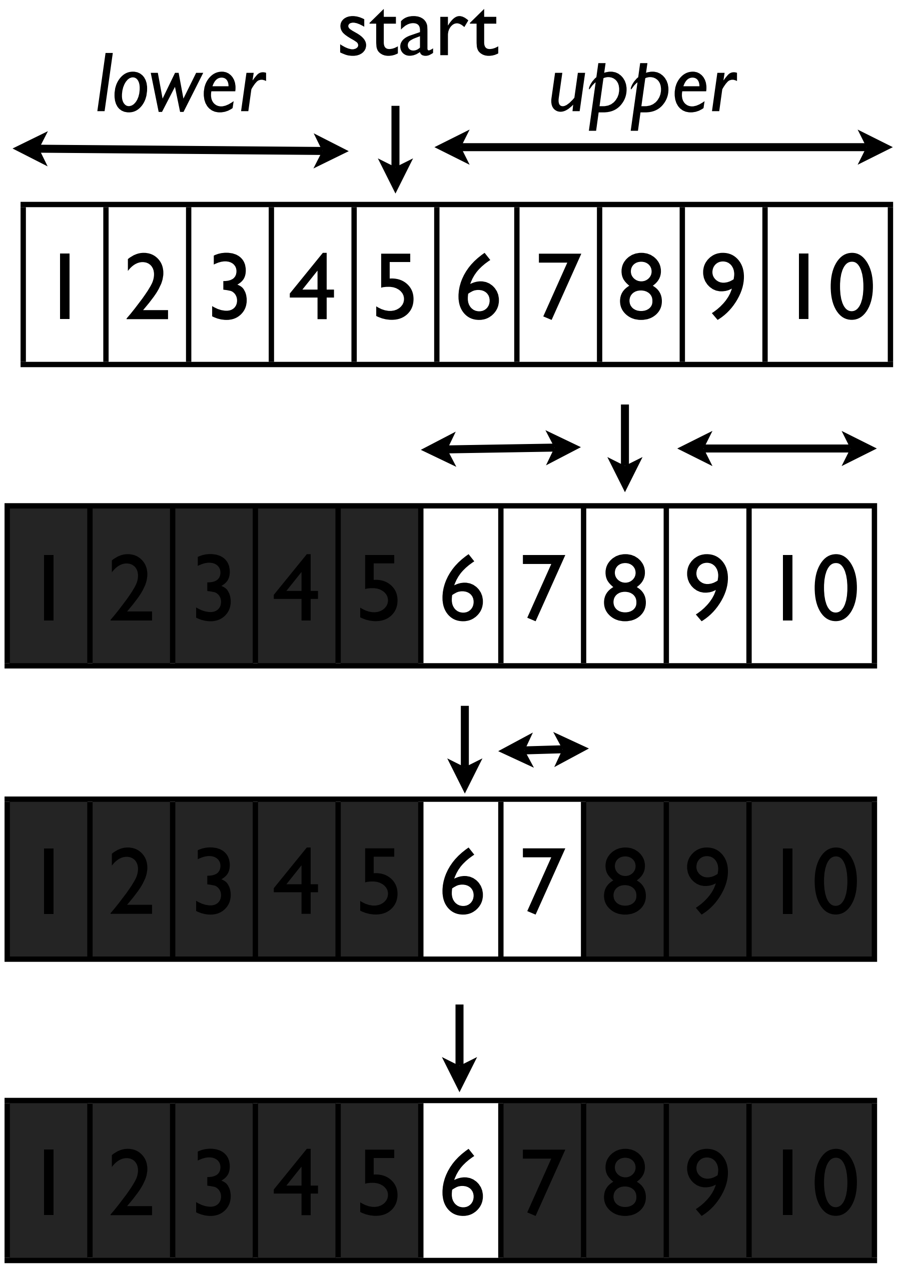


Where is the Base Case?

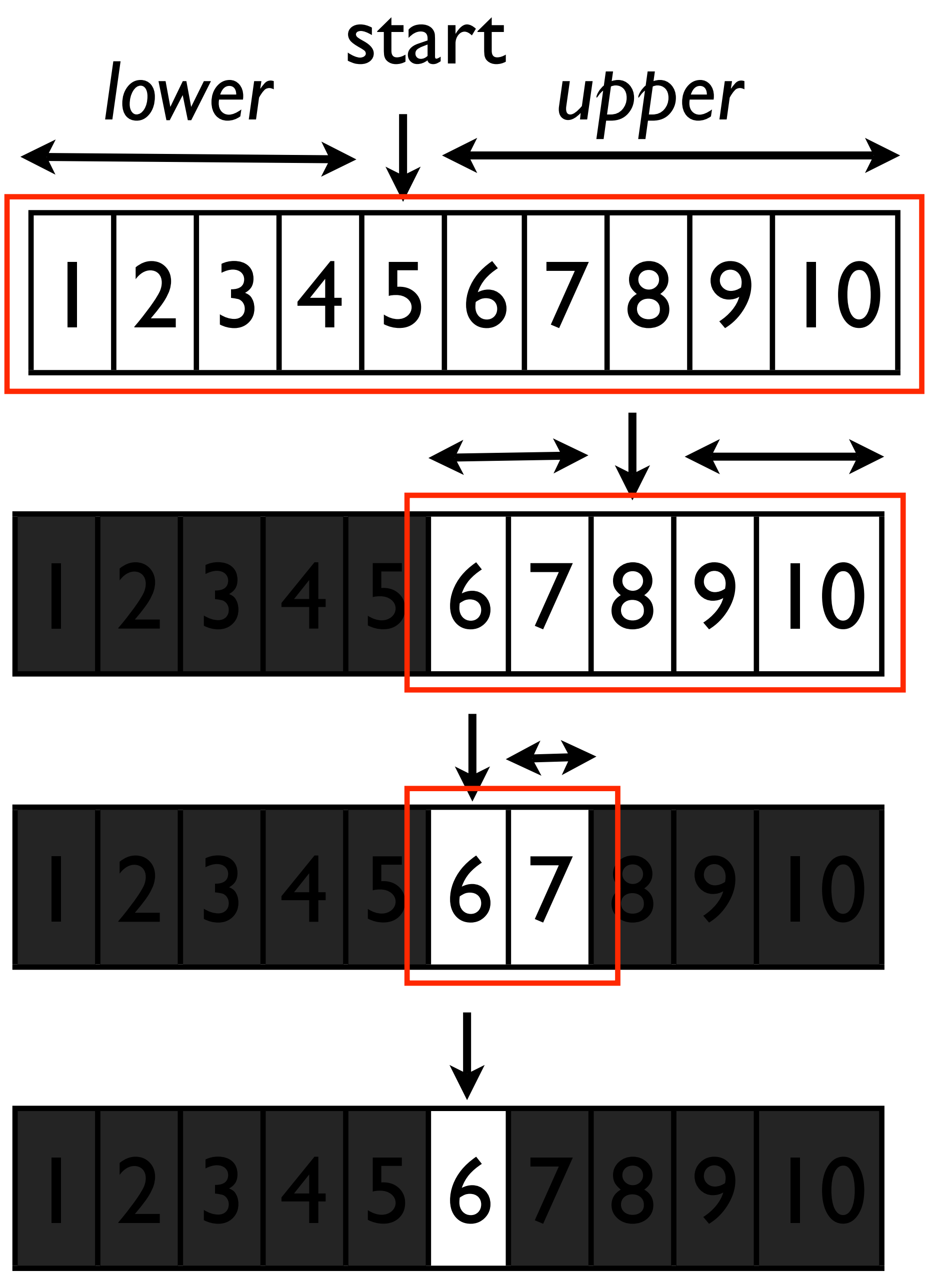


Check if the middle array element is the solution.

What is the Recursive Case?



What is the Recursive Case?



Binary Search Broken Down into smaller similar Problems

Convert the Binary Search into a Recursive Solution

```
int binarySearch(int array[], int numelems, int value)
{
   int first = 0,                  // First array element
       last = numelems - 1,        // Last array element
       middle,                     // Mid point of search
       position = -1;              // Position of search value
   bool found = false;             // Flag

   while (!found && first <= last)
   {
      middle = (first + last) / 2;      // Calculate mid point
      if (array[middle] == value)       // If value is found at mid
      {
         found = true;
         position = middle;
      }
      else if (array[middle] > value)   // If value is in lower half
         last = middle - 1;
      else
         first = middle + 1;            // If value is in upper half
   }
   return position;
}
```

Convert the Binary Search into a Recursive Solution

```
int binarySearch(int array[], int first, int last, int value)
{
```

Look in the Book!

```
}
```

Convert the Binary Search into a Recursive Solution

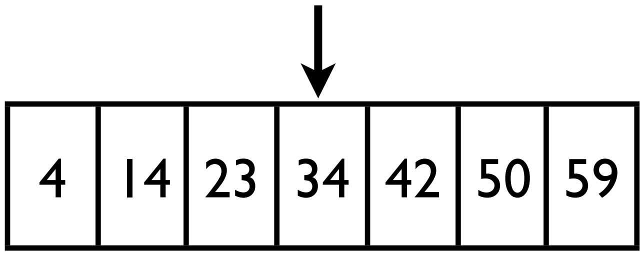
```
int binarySearch(int array[], int first, int last, int value)
{
  int middle; // Mid point of the search

  if(first > last)
    return -1;
  middle = (first + last) / 2;

  if (array[middle] == value)
    return middle;
  if (array[middle] < value)
    return binarySearch(array, middle+1, last, value);
  else
    return binarySearch(array, first, middle - 1, value);
}
```

Convert the Binary Search into a Recursive Solution (with tracing)

```
...

int stations[SIZE] = {4, 14, 23, 34, 42, 50, 59};

int query = 14;

int id;

id = binarySearch(stations, 0, SIZE-1, query);

...
```

```
$ dbx binSearch
(dbx) trace in binarySearch
(dbx) run
```

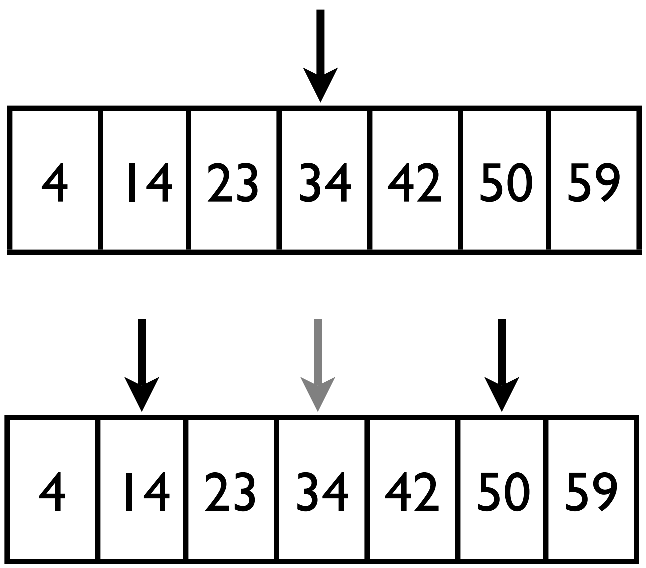
Foreshadowing to Binary Search Trees

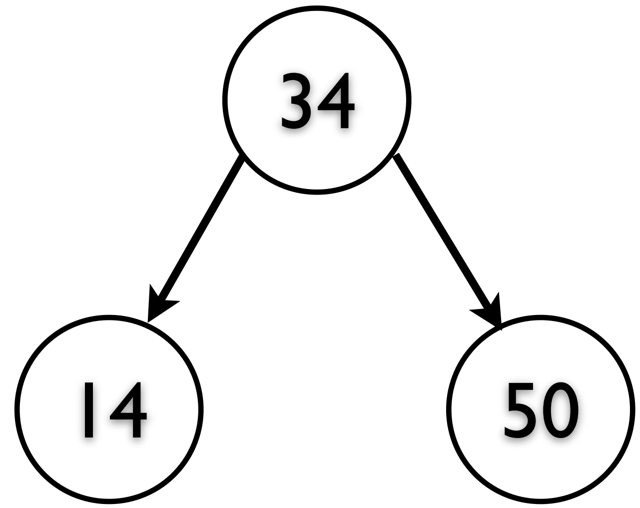


A ***tree*** is a *graph* data structure that is *directed* and *a-cyclical.*

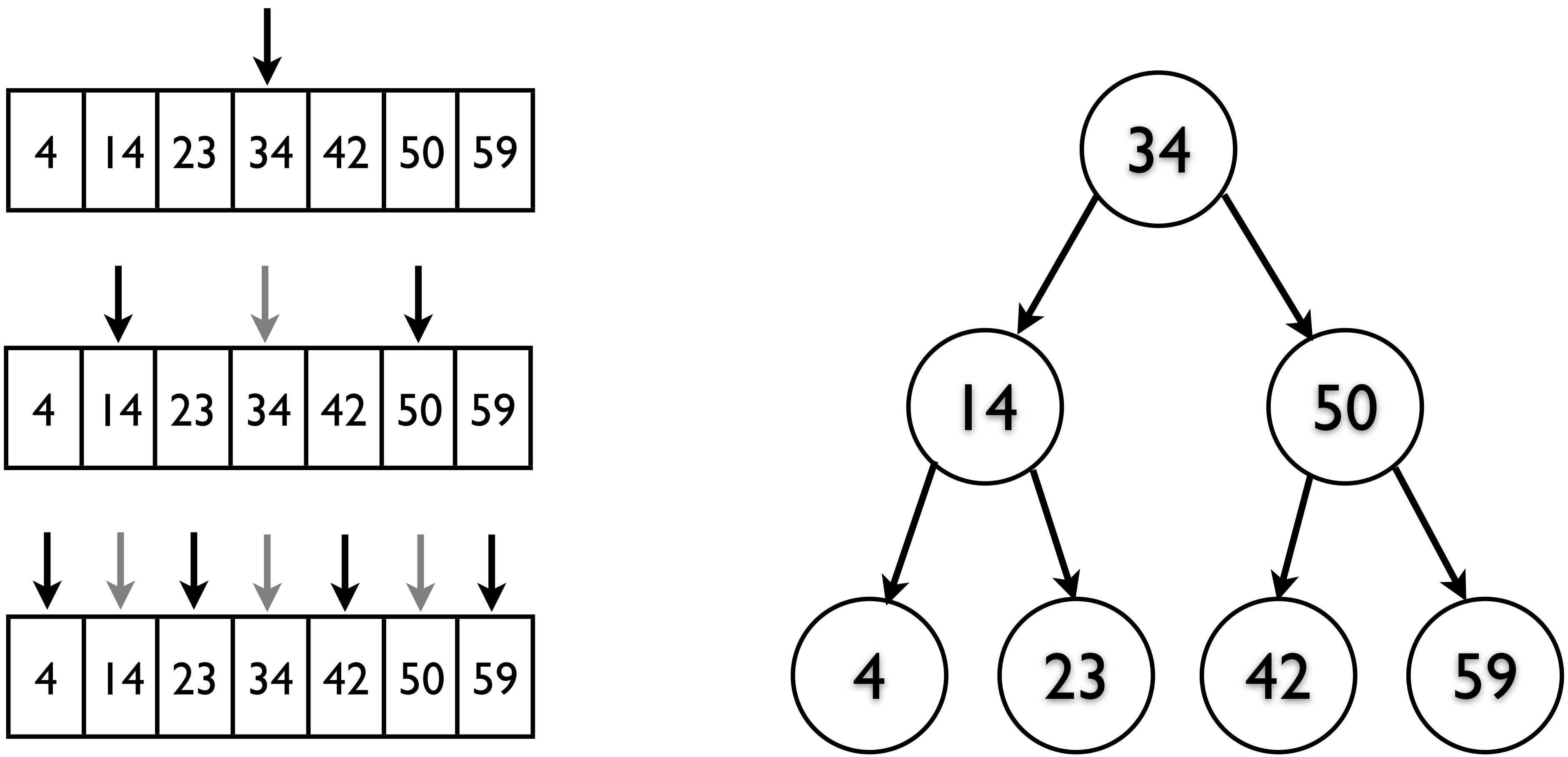
We will build a **tree** of the binary search out of the *middle* elements of every sub-array.

Foreshadowing to Binary Search Trees

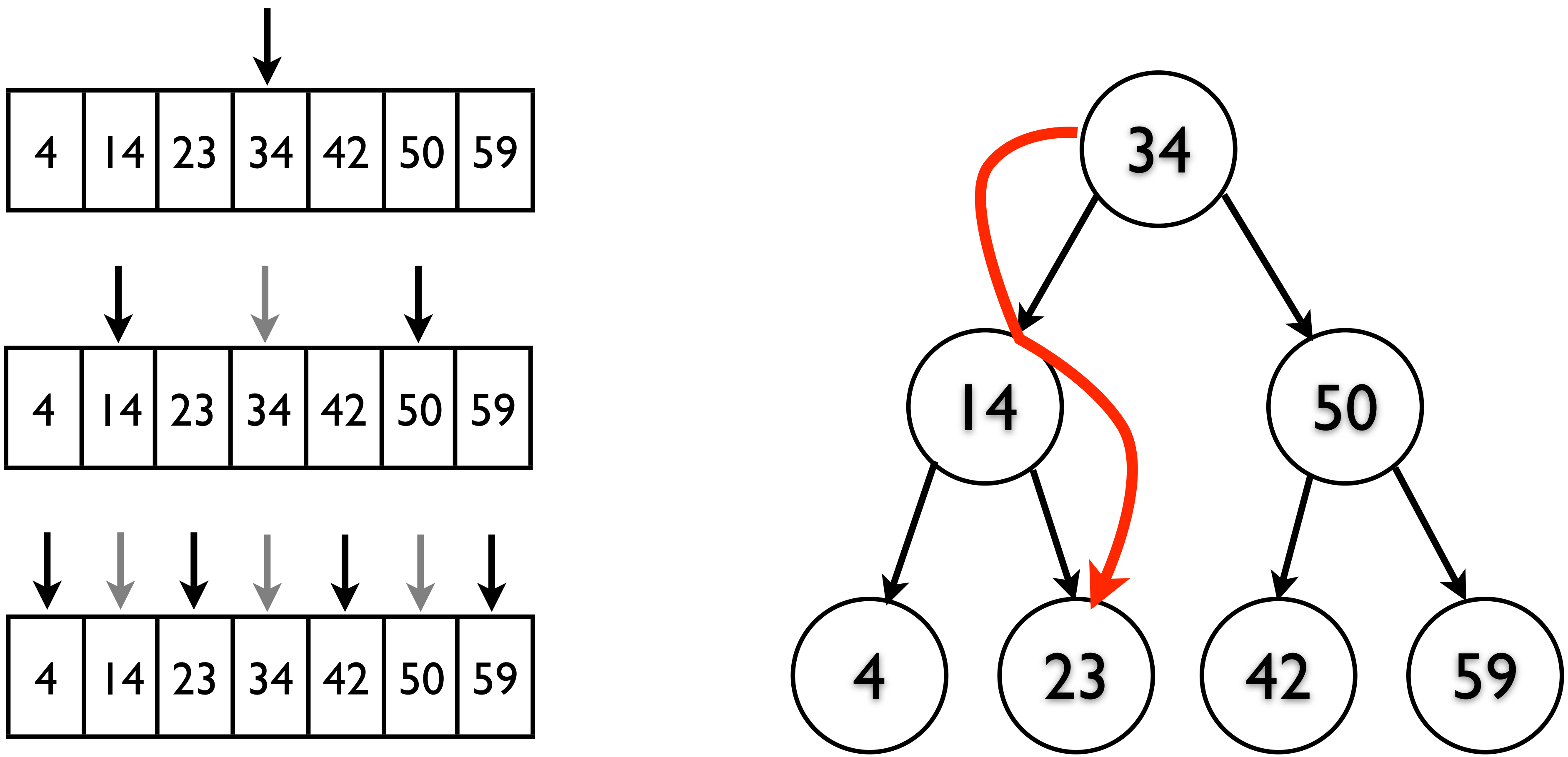




Foreshadowing to Binary Search Trees



Foreshadowing to Binary Search Trees



Searching for Element 23

QuickSort

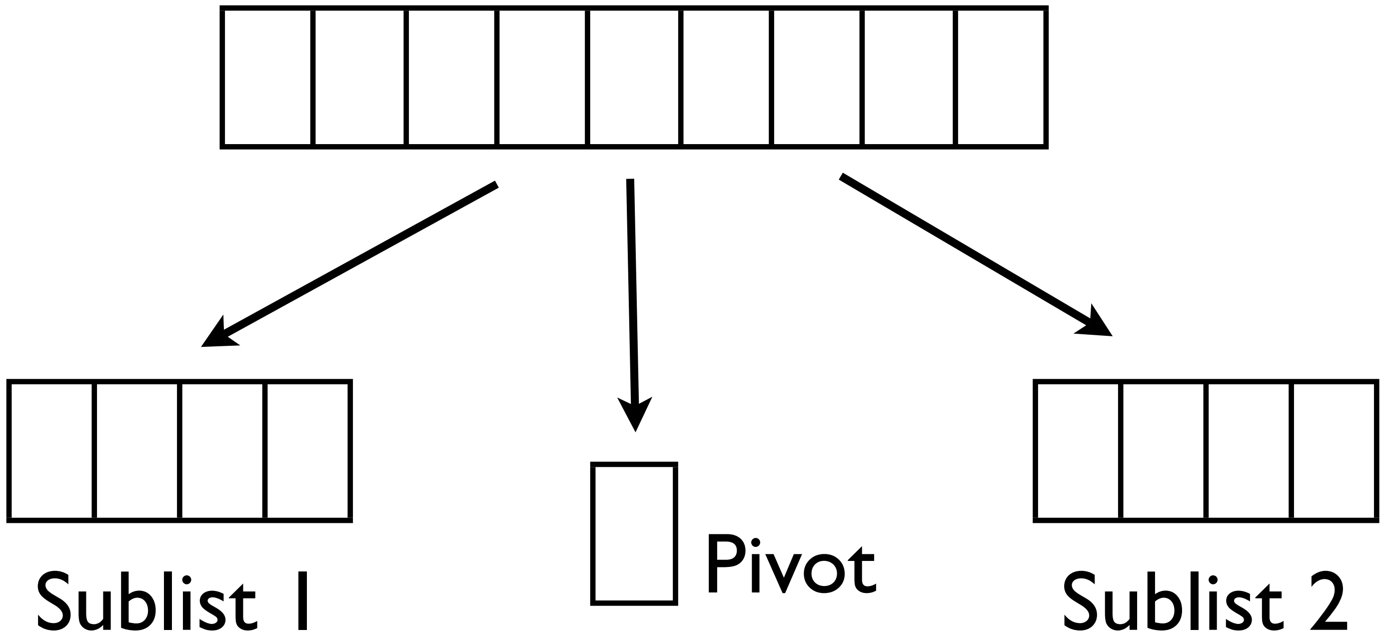
Developed in 1960 by C.A.R. Hoare (a British Computer Scientist).

Widely used Sorting Algorithm

Uses Recursion to Efficiently sort a list

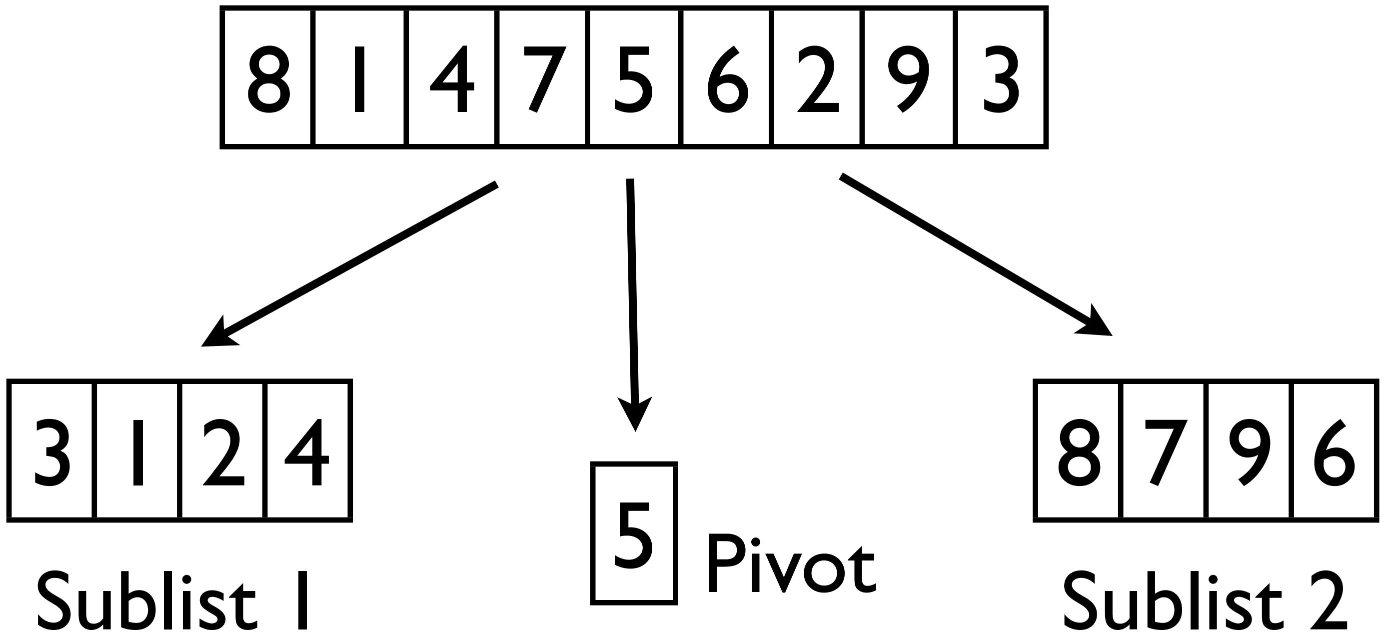
It uses a ***divide-and-conquer*** technique.

QuickSort - Divide an Conquer



The Divide-And-Conquer approach *partitions* the list into 2 **Sublists** about a **Pivot** point.

QuickSort - Divide an Conquer



Sublist 1 contain values that are less than the Pivot *value*.

Sublist 2 contain values that are greater than the Pivot *value*.

Note: Sublists are not necessarily sorted.

Now, guess what happens?

QuickSort - Divide an Conquer

Sublist 1

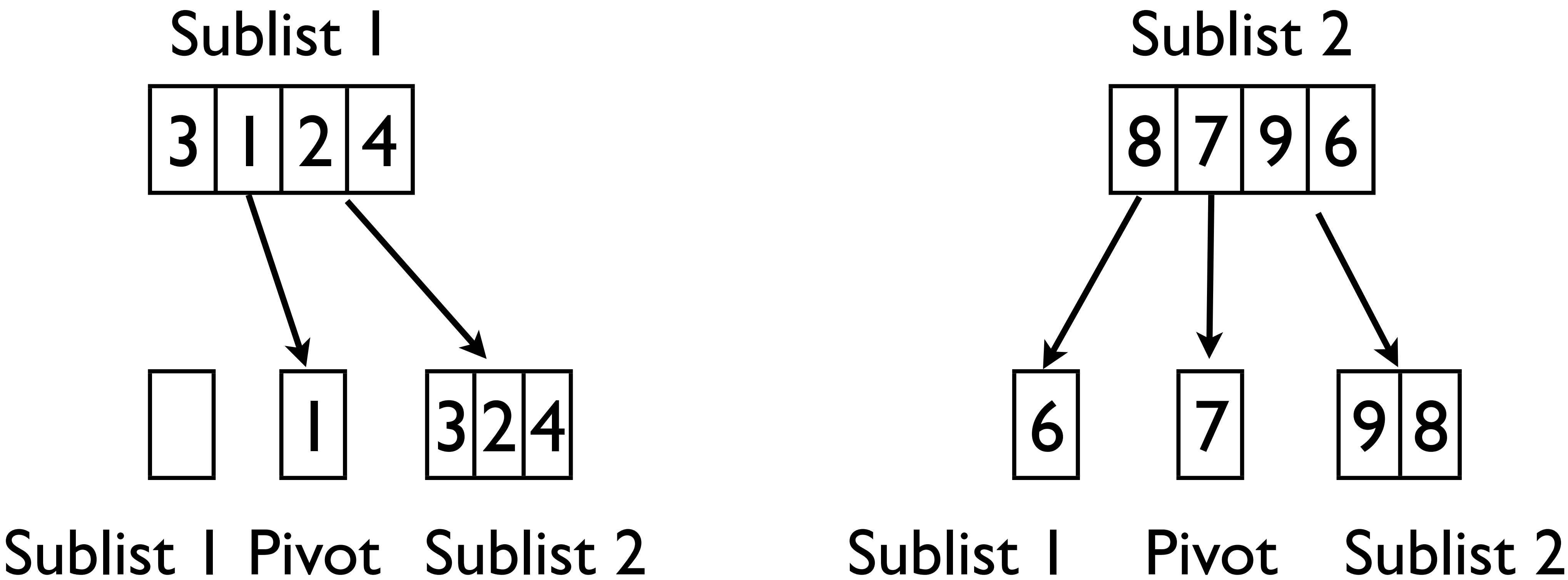
3	1	2	4

Sublist 2

8	7	9	6

QuickSort is recursively called on both Sublists!

QuickSort - Divide an Conquer



QuickSort is recursively called on both Sublists to partition them.

Notice: Partition is based on the *value* of the Pivot.

Thus Sublists that result from the Partition may differ in **size**.

QuickSort - the Algorithm

```
void quickSort(int set[], int start, int end)
{
   int pivotPoint;

   if (start < end)
   {
      // Get the pivot point.
      pivotPoint = partition(set, start, end);
      // Sort the first sub list.
      quickSort(set, start, pivotPoint - 1);
      // Sort the second sub list.
      quickSort(set, pivotPoint + 1, end);
   }
}
```

Partition re-arranges the two lists

```
int partition(int set[], int start, int end)
{
   int pivotValue, pivotIndex, mid;

   mid = (start + end) / 2;
   swap(set[start], set[mid]);
   pivotIndex = start;
   pivotValue = set[start];
   for (int scan = start + 1; scan <= end; scan++)
   {
      if (set[scan] < pivotValue)
      {
         pivotIndex++;
         swap(set[pivotIndex], set[scan]);
      }
   }
   swap(set[start], set[pivotIndex]);
   return pivotIndex;
}
```

```
void swap(int &value1, int &value2)
{
   int temp = value1;

   value1 = value2;
   value2 = temp;
}
```

More than one way to Partition a List of numbers!

8	1	4	7	5	6	2	9	3

```
int partition(int set[], int start, int end)
{
   int pivotValue, pivotIndex, mid;

   mid = (start + end) / 2;
   swap(set[start], set[mid]);
   pivotIndex = start;
   pivotValue = set[start];
   for (int scan = start + 1; scan <= end; scan++)
   {
      if (set[scan] < pivotValue)
      {
         pivotIndex++;
         swap(set[pivotIndex], set[scan]);
      }
   }
   swap(set[start], set[pivotIndex]);
   return pivotIndex;
}
```

What is the resultant array after partition?

```
void swap(int &value1, int &value2)
{
   int temp = value1;

   value1 = value2;
   value2 = temp;
}
```

What are the intermediate arrays?

```
void quickSort(int set[], int start, int end)
{
    int pivotPoint;

    if (start < end)
    {
        // DISPLAY the ARRAY HERE //
        // Get the pivot point.
        pivotPoint = partition(set, start, end);
        // Sort the first sub list.
        quickSort(set, start, pivotPoint - 1);
        // Sort the second sub list.
        quickSort(set, pivotPoint + 1, end);
    }
}
```

Start	8	1	4	7	5	6	2	9	3

Try it out

```
$ ./qs
```

Does your output match this output?

QuickSort Efficiency

What is the worse case for QuickSort?

If the Pivot is always the smallest or largest value in the list

End up scanning the entire list n times: n operations.

Turns out, the AVERAGE running time is n log(n) operations

$n \log(n) < n^2$!

And now for something different...

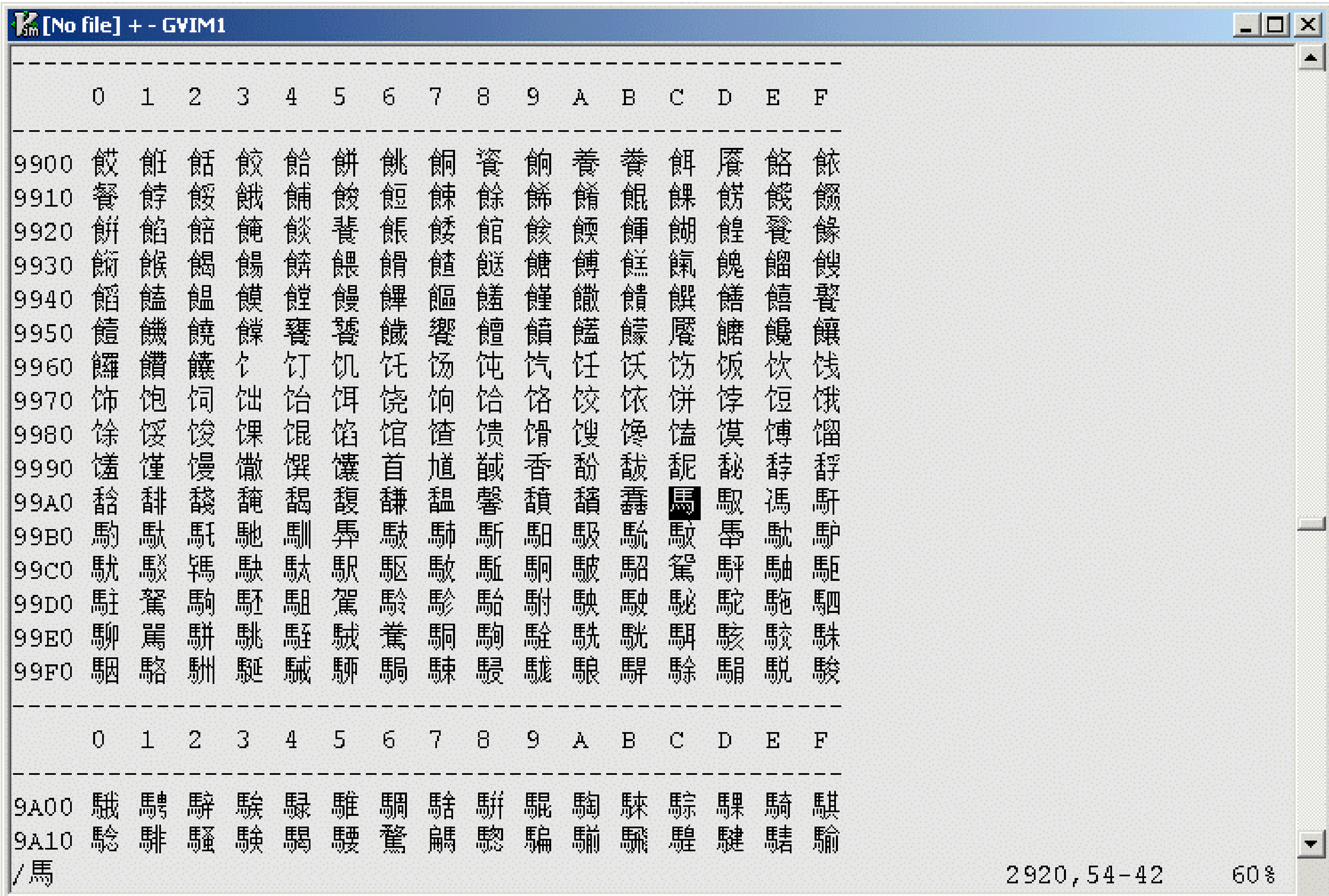
Strings

Strings are comprised of Characters (8 bit numbers)

Dec	Hx	Oct	Char		Dec	Hx	Oct	Html	Chr	Dec	Hx	Oct	Html	Chr	Dec	Hx	Oct	Html	Chr
0	0	000	NUL	(null)	32	20	040		Space	64	40	100	@	@	96	60	140	`	`
1	1	001	SOH	(start of heading)	33	21	041	!	!	65	41	101	A	A	97	61	141	a	a
2	2	002	STX	(start of text)	34	22	042	"	"	66	42	102	B	B	98	62	142	b	b
3	3	003	ETX	(end of text)	35	23	043	#	#	67	43	103	C	C	99	63	143	c	c
4	4	004	EOT	(end of transmission)	36	24	044	$	$	68	44	104	D	D	100	64	144	d	d
5	5	005	ENQ	(enquiry)	37	25	045	%	%	69	45	105	E	E	101	65	145	e	e
6	6	006	ACK	(acknowledge)	38	26	046	&	&	70	46	106	F	F	102	66	146	f	f
7	7	007	BEL	(bell)	39	27	047	'	'	71	47	107	G	G	103	67	147	g	g
8	8	010	BS	(backspace)	40	28	050	(	(	72	48	110	H	H	104	68	150	h	h
9	9	011	TAB	(horizontal tab)	41	29	051	)	)	73	49	111	I	I	105	69	151	i	i
10	A	012	LF	(NL line feed, new line)	42	2A	052	*	*	74	4A	112	J	J	106	6A	152	j	j
11	B	013	VT	(vertical tab)	43	2B	053	+	+	75	4B	113	K	K	107	6B	153	k	k
12	C	014	FF	(NP form feed, new page)	44	2C	054	,	,	76	4C	114	L	L	108	6C	154	l	l
13	D	015	CR	(carriage return)	45	2D	055	-	-	77	4D	115	M	M	109	6D	155	m	m
14	E	016	SO	(shift out)	46	2E	056	.	.	78	4E	116	N	N	110	6E	156	n	n
15	F	017	SI	(shift in)	47	2F	057	/	/	79	4F	117	O	O	111	6F	157	o	o
16	10	020	DLE	(data link escape)	48	30	060	0	0	80	50	120	P	P	112	70	160	p	p
17	11	021	DC1	(device control 1)	49	31	061	1	1	81	51	121	Q	Q	113	71	161	q	q
18	12	022	DC2	(device control 2)	50	32	062	2	2	82	52	122	R	R	114	72	162	r	r
19	13	023	DC3	(device control 3)	51	33	063	3	3	83	53	123	S	S	115	73	163	s	s
20	14	024	DC4	(device control 4)	52	34	064	4	4	84	54	124	T	T	116	74	164	t	t
21	15	025	NAK	(negative acknowledge)	53	35	065	5	5	85	55	125	U	U	117	75	165	u	u
22	16	026	SYN	(synchronous idle)	54	36	066	6	6	86	56	126	V	V	118	76	166	v	v
23	17	027	ETB	(end of trans. block)	55	37	067	7	7	87	57	127	W	W	119	77	167	w	w
24	18	030	CAN	(cancel)	56	38	070	8	8	88	58	130	X	X	120	78	170	x	x
25	19	031	EM	(end of medium)	57	39	071	9	9	89	59	131	Y	Y	121	79	171	y	y
26	1A	032	SUB	(substitute)	58	3A	072	:	:	90	5A	132	Z	Z	122	7A	172	z	z
27	1B	033	ESC	(escape)	59	3B	073	;	;	91	5B	133	[	[	123	7B	173	{	{
28	1C	034	FS	(file separator)	60	3C	074	<	<	92	5C	134	\	\	124	7C	174			|
29	1D	035	GS	(group separator)	61	3D	075	=	=	93	5D	135	]	]	125	7D	175	}	}
30	1E	036	RS	(record separator)	62	3E	076	>	>	94	5E	136	^	^	126	7E	176	~	~
31	1F	037	US	(unit separator)	63	3F	077	?	?	95	5F	137	_	_	127	7F	177	&#127;	DEL

What's wrong with using just 8 bits?

Unicode (various mappings - up to 32 bits)



ASCII Character Testing

```
#include <cctype>



int isalpha(char n);

int isalnum(char n);

int isdigit(char n);

int islower(char n);

int isprint(char n);

int ispunct(char n);

int isupper(char n);

int isspace(char n);
```

Character Testing

```
#include <cctype>
```

`int isalpha(char n);`	Alphabet
`int isalnum(char n);`	Alphabet or Digit
`int isdigit(char n);`	0 through 9
`int islower(char n);`	lowercase
`int isprint(char n);`	printable
`int ispunct(char n);`	punctuation
`int isupper(char n);`	uppercase
`int isspace(char n);`	space

Character Testing

```
#include <cctype>
```

`int isalpha(char n);`	Alphabet
`int isalnum(char n);`	Alphabet or Digit
`int isdigit(char n);`	0 through 9
`int islower(char n);`	lowercase
`int isprint(char n);`	printable
`int ispunct(char n);`	punctuation
`int isupper(char n);`	uppercase
`int isspace(char n);`	space

Returns True or False how?

Character Testing

```
#include <cctype>
```

```
int isalpha(char n);
```
Alphabet

```
int isalnum(char n);
```
Alphabet or Digit

```
int isdigit(char n);
```
0 through 9

```
int islower(char n);
```
lowercase

```
int isprint(char n);
```
printable

```
int ispunct(char n);
```
punctuation

```
int isupper(char n);
```
uppercase

```
int isspace(char n);
```
space

Returns True
or False
how?

0 is False

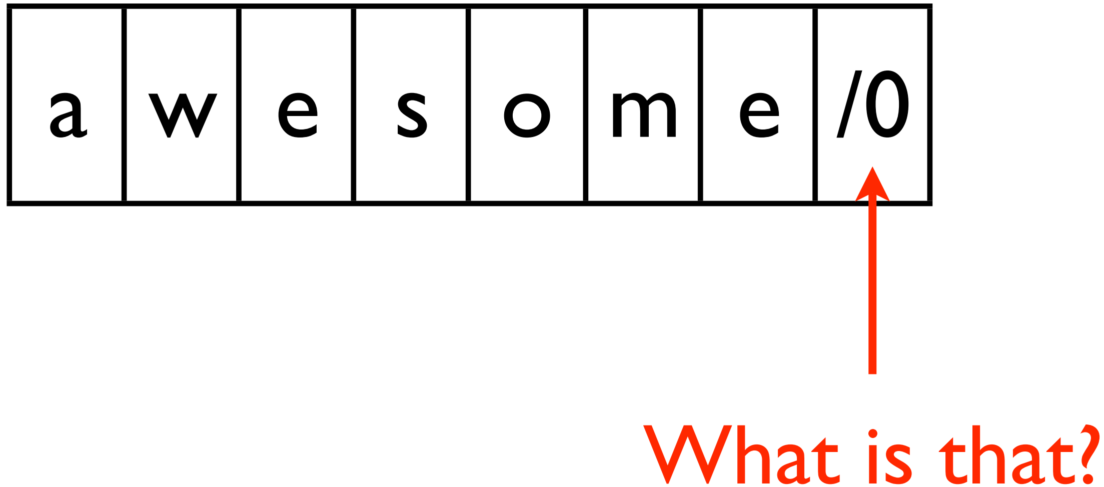
> 0 is True

Case Conversion

```
char toupper(char c);

char tolower(char c);
```

```
cout << toupper(tolower(toupper(tolower(toupper('c')))));
```

Strings as an collection of characters!



Strings as an collection of characters!

a	w	e	s	o	m	e	/0

How the computer views it!

97	119	101	115	111	109	101	0

Store Strings in Arrays

a	w	e	s	o	m	e	/0

```
char myString[8];
cin >> myString;  // what is the problem here?
```

Store Strings in Arrays

a	w	e	s	o	m	e	/0

```
char myString[8];

cin >> myString;  // what is the problem here?

cin.getline(myString, 8);
```

Summary

Recursive Implementation of Binary Search

QuickSort (a Recursive Sorting Algorithm)

String Library Review

READINGS: 19.6, and 19.8 - 19.10, Chapter 10.1 - 10.5