Recursive Maximum

Iterative version

- Have a array or vector called A. Want to find the maximum element:

```
biggest = A[0]
for (size_t p = 0; p < A.size(); p++)
  if (A[p] > biggest)
    biggest = A[p];
/* After the loop, we know biggest is
the maximum element in A.  */
```

Recursive version

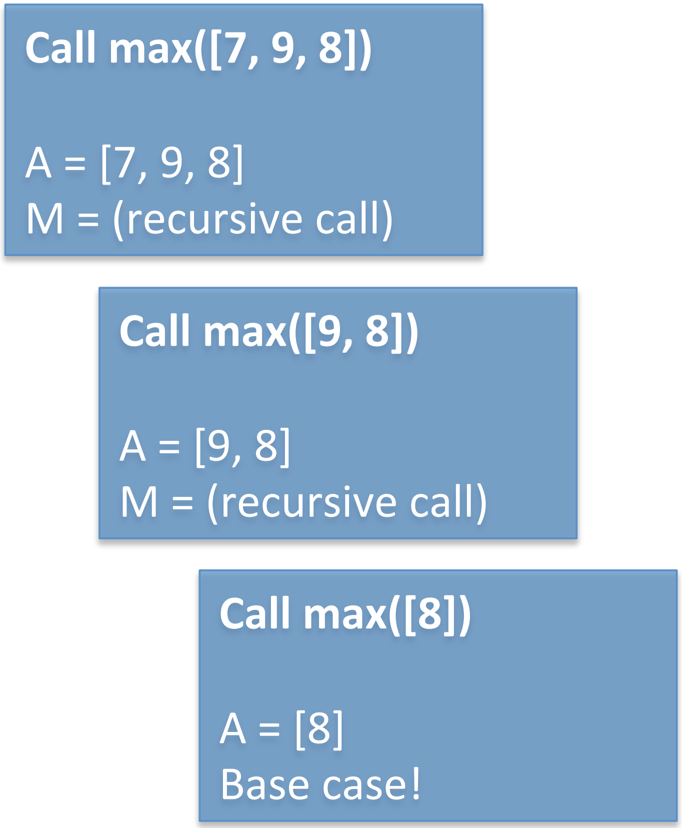
- Base case: What is the smallest size array I would ever want to find the maximum element in?
- Recursive case:
  - Suppose you have an array A (with >1 element).
  - How can I describe finding the maximum element as involving ***finding the maximum element of a smaller sized array?***
  - Hint: Suppose my array has 5 elements. My best friend knows how to find the largest value in an array, but only for 4 elements. How can I use him to solve my problem?

Recursive version

- max(A)
- Base case: If A.size() == 1, return A[0]
- Recursive case: If A.size() > 1:
  - Find the maximum element in A[1:] (whole array except A[0])
    - call it M
  - If M > A[0]: return M
  - Else: return A[0]

- max(A)
- Base case: If A.size() == 1, return A[0]
- Recursive case: If A.size() > 1:
  - Find the maximum element in A[1:] (whole array except A[0])
    - call it M
  - If M > A[0]: return M
  - Else: return A[0]

A = [7, 9, 8]



Call max([7, 9, 8])

A = [7, 9, 8]
M = (recursive call)

Call max([9, 8])

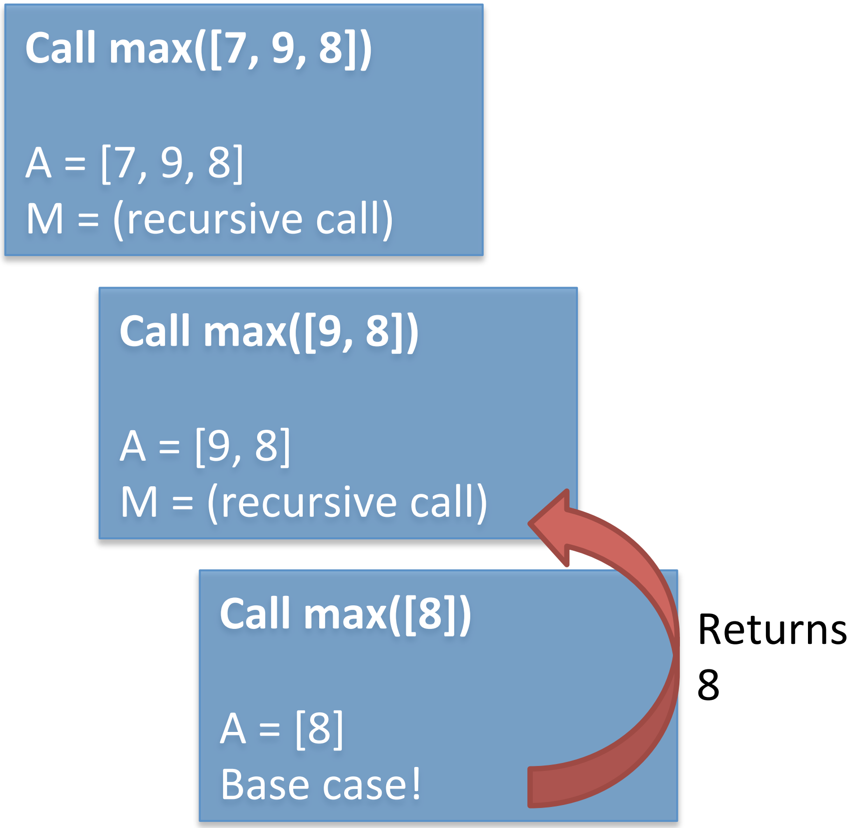
A = [9, 8]
M = (recursive call)

Call max([8])

A = [8]
Base case!

A = [7, 9, 8]

- max(A)
- Base case: If A.size() == 1, return A[0]
- Recursive case: If A.size() > 1:
  - Find the maximum element in A[1:] (whole array except A[0])
    - call it M
  - If M > A[0]: return M
  - Else: return A[0]



Call max([7, 9, 8])

A = [7, 9, 8]
M = (recursive call)

Call max([9, 8])

A = [9, 8]
M = (recursive call)

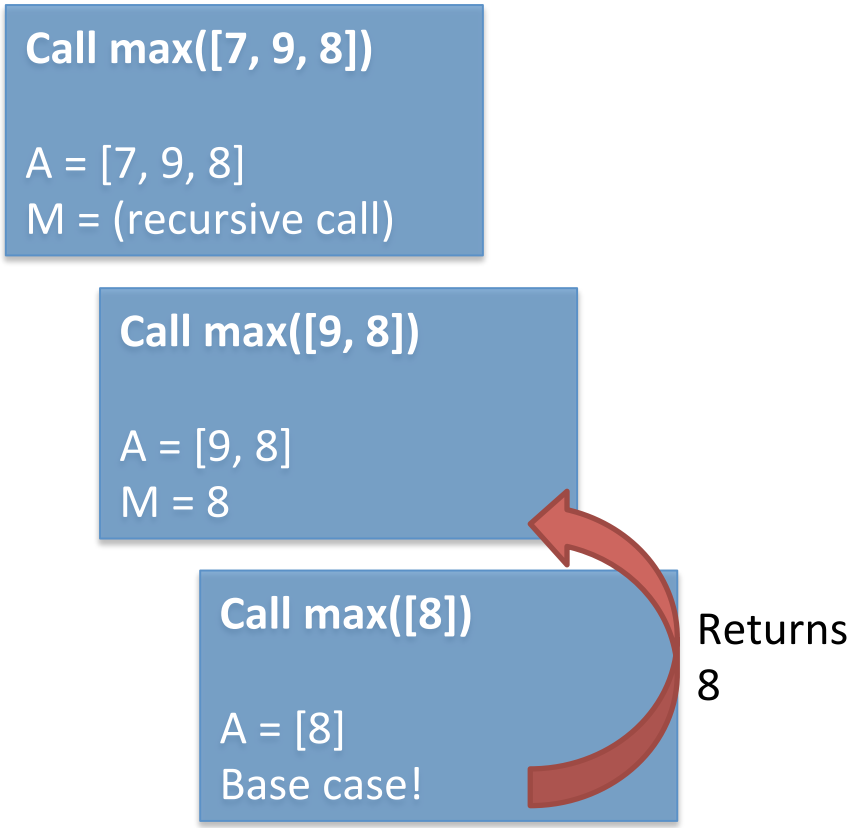
Call max([8])

A = [8]
Base case!

Returns
8

- max(A)
- Base case: If A.size() == 1, return A[0]
- Recursive case: If A.size() > 1:
  - Find the maximum element in A[1:] (whole array except A[0])
    - call it M
  - If M > A[0]: return M
  - Else: return A[0]

A = [7, 9, 8]



Call max([7, 9, 8])

A = [7, 9, 8]
M = (recursive call)

Call max([9, 8])

A = [9, 8]
M = 8

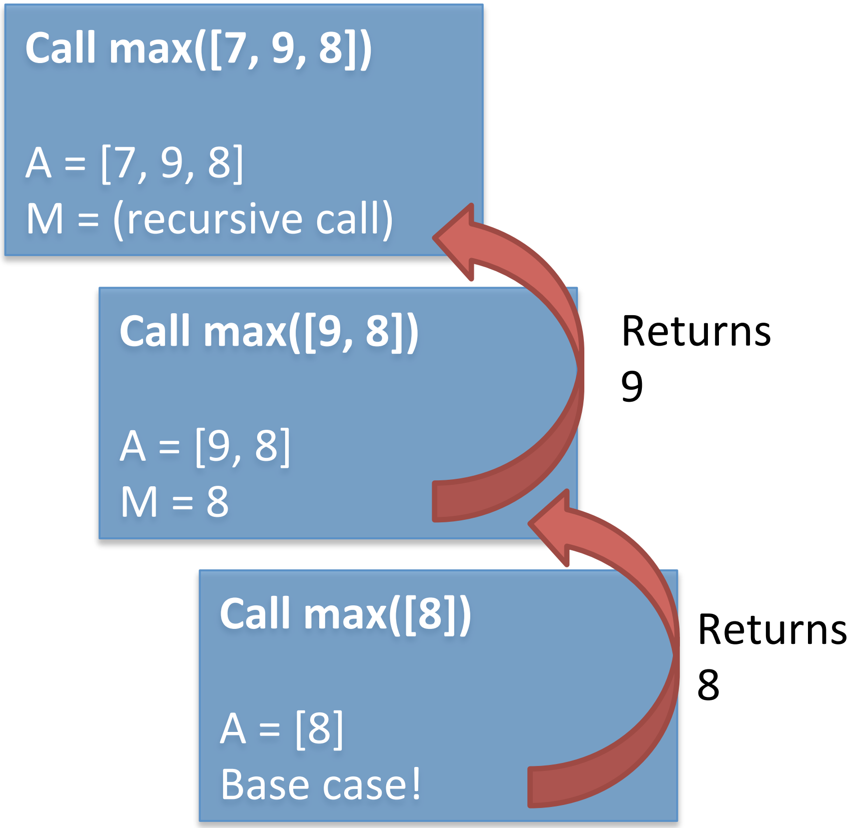
Call max([8])

A = [8]
Base case!

Returns
8

- max(A)
- Base case: If A.size() == 1, return A[0]
- Recursive case: If A.size() > 1:
  - Find the maximum element in A[1:] (whole array except A[0])
    - call it M
  - If M > A[0]: return M
  - Else: return A[0]

A = [7, 9, 8]



Call max([7, 9, 8])

A = [7, 9, 8]
M = (recursive call)

Call max([9, 8])

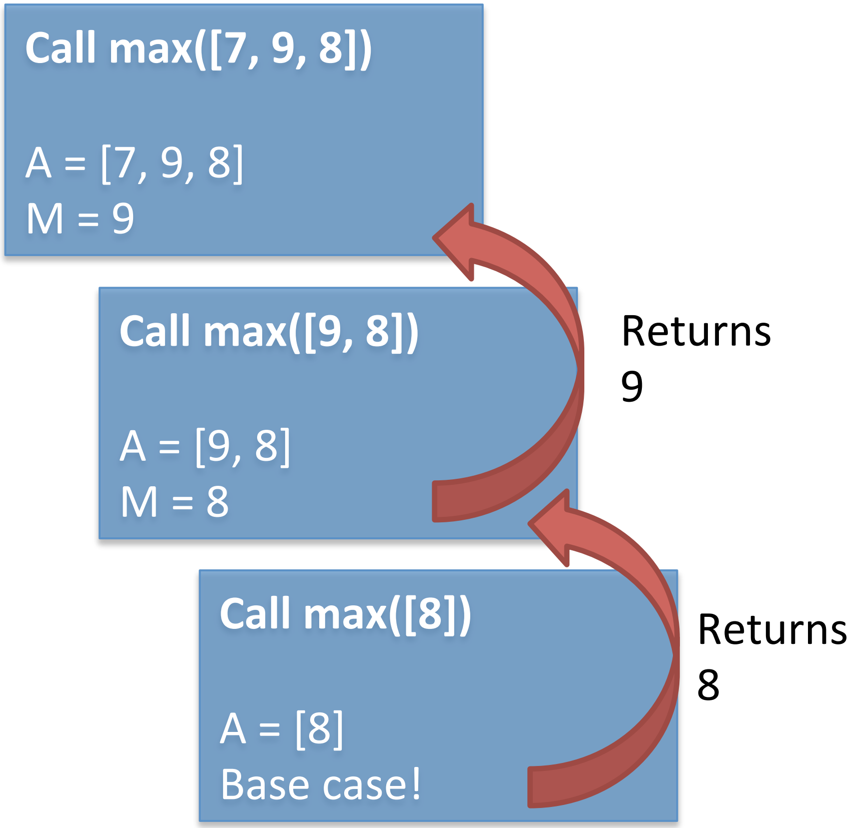
A = [9, 8]
M = 8

Call max([8])

A = [8]
Base case!

- max(A)
- Base case: If A.size() == 1, return A[0]
- Recursive case: If A.size() > 1:
  - Find the maximum element in A[1:] (whole array except A[0])
    - call it M
  - If M > A[0]: return M
  - Else: return A[0]

A = [7, 9, 8]



Call max([7, 9, 8])

A = [7, 9, 8]
M = 9

Call max([9, 8])

A = [9, 8]
M = 8

Call max([8])

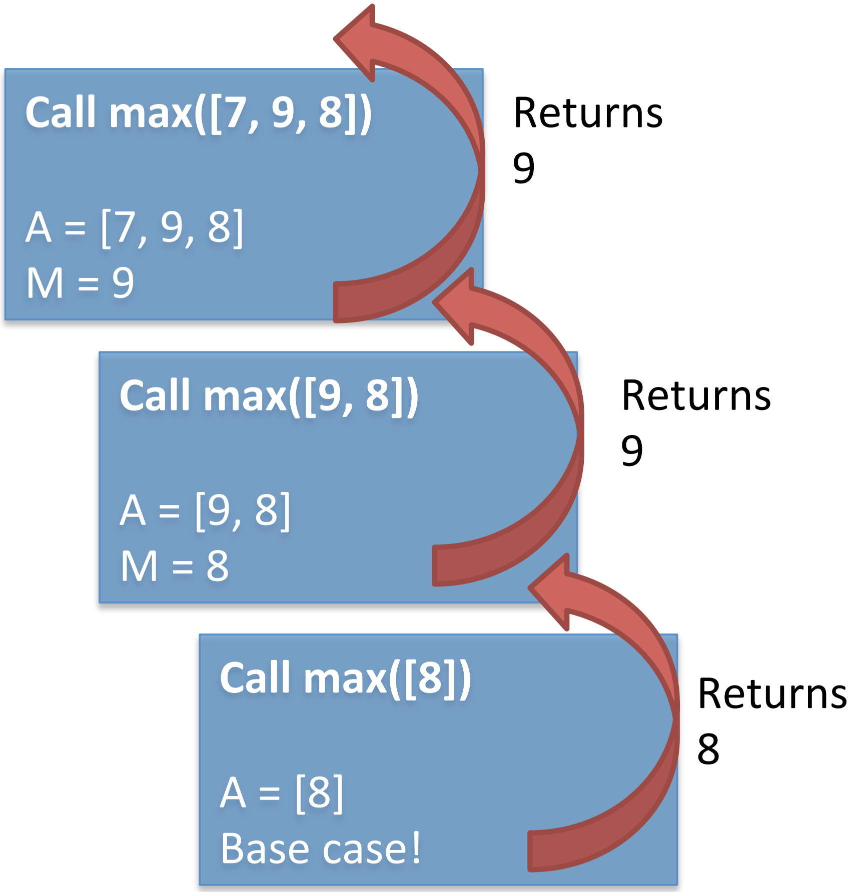
A = [8]
Base case!

Returns 9

Returns 8

- max(A)
- Base case: If A.size() == 1, return A[0]
- Recursive case: If A.size() > 1:
  - Find the maximum element in A[1:] (whole array except A[0])
    - call it M
  - If M > A[0]: return M
  - Else: return A[0]

A = [7, 9, 8]



Call max([7, 9, 8])

A = [7, 9, 8]
M = 9

Returns 9

Call max([9, 8])

A = [9, 8]
M = 8

Returns 9

Call max([8])

A = [8]
Base case!

Returns 8

C++ recursive version

- C++ doesn't let you take slices of arrays (also inefficient).
- Notice that our slices always involving chopping off the first element in the array; i.e, A[0]
  - [7, 9, 8] -> [9, 8] -> [8]
- How can we simulate an array slice without actually doing the slicing?
  - Hint: Imagine reading a textbook (lol). When you finish reading a page, you don't rip it out of the book, yet you want to be able to return to that place in the book later to study more (rofl). How do you solve this conundrum?

C++ recursive version

- Use a integer variable "bookmark" to save your spot in the array.
- When we make a recursive call, instead of passing an updated array A (like the Python version), we will pass an updated bookmark.
- Our function will now be max(A, low)
  - low (the bookmark) represents the index of the bookmark in the array: everything before the bookmark is already read, everything afterwards is unread.

Recursive C++ version

- max(A, low)
- Base case: ???
- Recursive case:
  - – Find the maximum element in ???
    - call it M
  - – if ???: return M
  - – else: return ???

- Where does the bookmark start?

Recursive C++ version

- max(A, low)
- Base case: if low == A.size() - 1
- Recursive case:
  - Find the maximum element in everything after A[low]
    - M = max(A, low + 1)
  - if M > A[low]: return M
  - else: return A[low]

- Initial call should be max(A, 0)

Binary Search

Phonebook

- Like linear search, binary search finds whether a certain item (the key) is in an array or vector.
- Binary search only works on ***sorted*** arrays or vectors.
  - Binary search takes advantage of the array being sorted to make the search much faster.

key = 33

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
6	13	14	25	33	43	51	53	64	72	84	93	95	96	97

key = 33

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
6	13	14	25	33	43	51	53	64	72	84	93	95	96	97

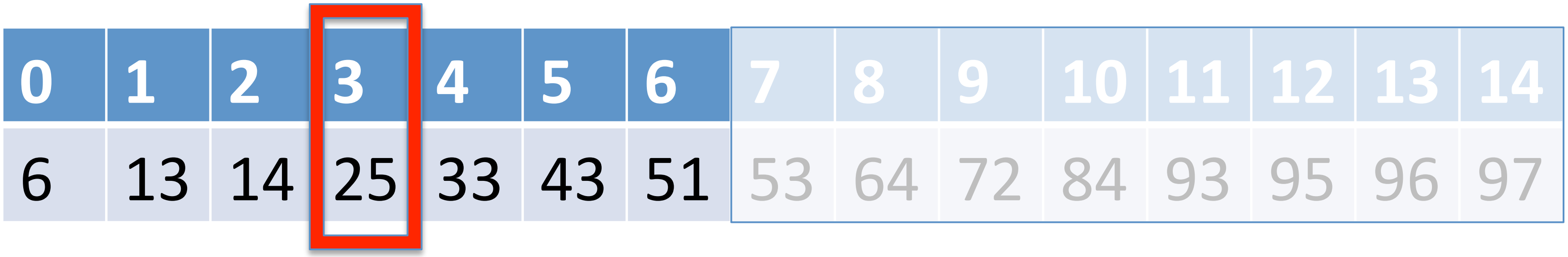
key = 33

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
6	13	14	25	33	43	51	53	64	72	84	93	95	96	97

key = 33

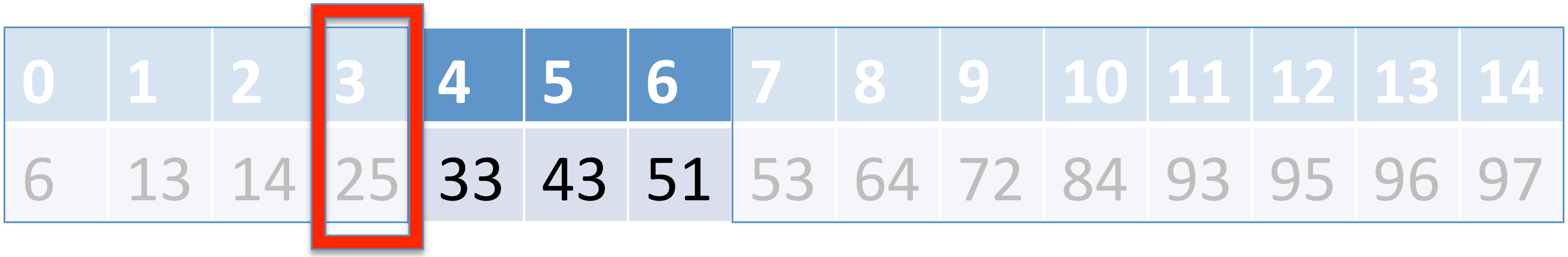
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
6	13	14	25	33	43	51	53	64	72	84	93	95	96	97

key = 33



0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
6	13	14	25	33	43	51	53	64	72	84	93	95	96	97

key = 33

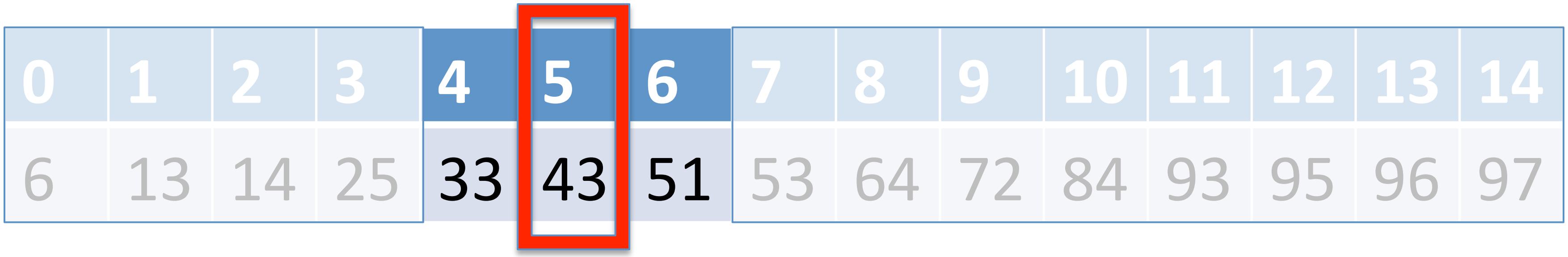


0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
6	13	14	25	33	43	51	53	64	72	84	93	95	96	97

key = 33

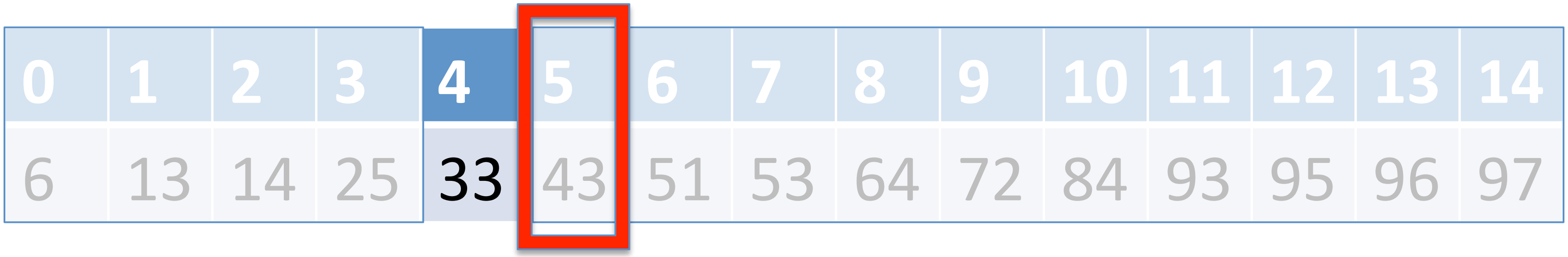
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
6	13	14	25	33	43	51	53	64	72	84	93	95	96	97

key = 33



0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
6	13	14	25	33	43	51	53	64	72	84	93	95	96	97

key = 33



0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
6	13	14	25	33	43	51	53	64	72	84	93	95	96	97

key = 33

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
6	13	14	25	33	43	51	53	64	72	84	93	95	96	97

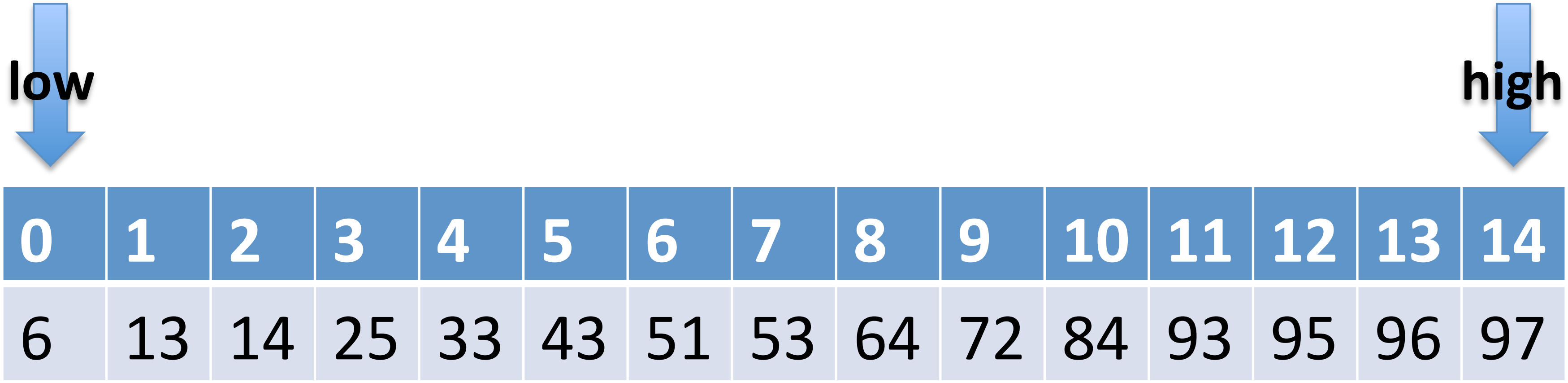
key = 33
Found! (return 4)

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
6	13	14	25	33	43	51	53	64	72	84	93	95	96	97

- Three variables that do most of the work:
  - low: the smallest index that could possibly contain the key.
  - high: the largest index that could possibly contain the key.
  - mid: the midpoint of the two indices.

- If low > high, we know the item is not found (stop).
- If array[mid] == key, item is found (stop).
- If array[mid] > key, repeat algorithm with only the left half of the array.
- If array[mid] < key, repeat algorithm with only the right half of the array.

key = 33

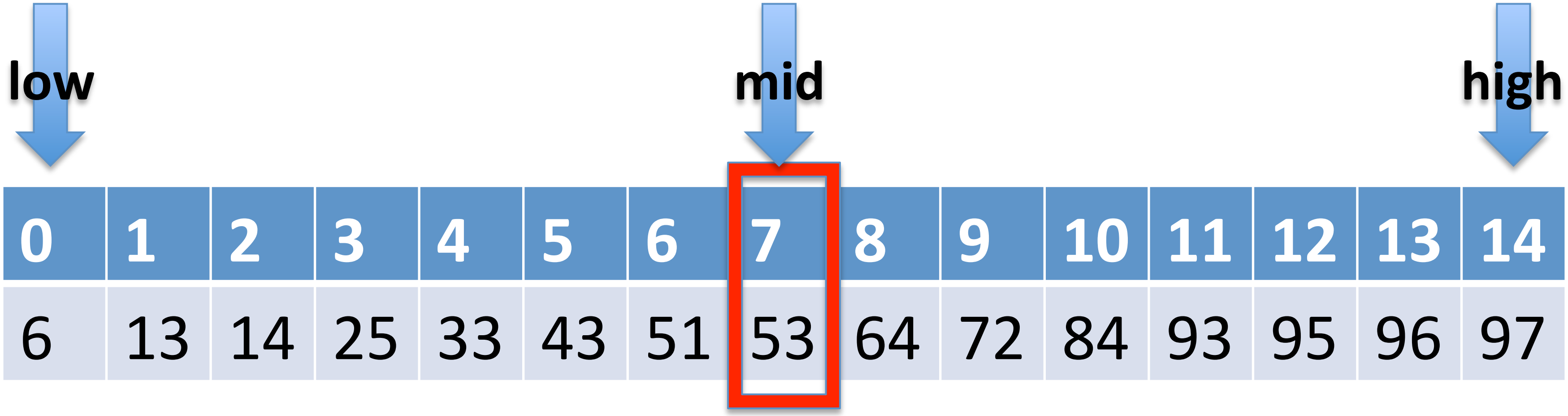


low

high

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
6	13	14	25	33	43	51	53	64	72	84	93	95	96	97

key = 33



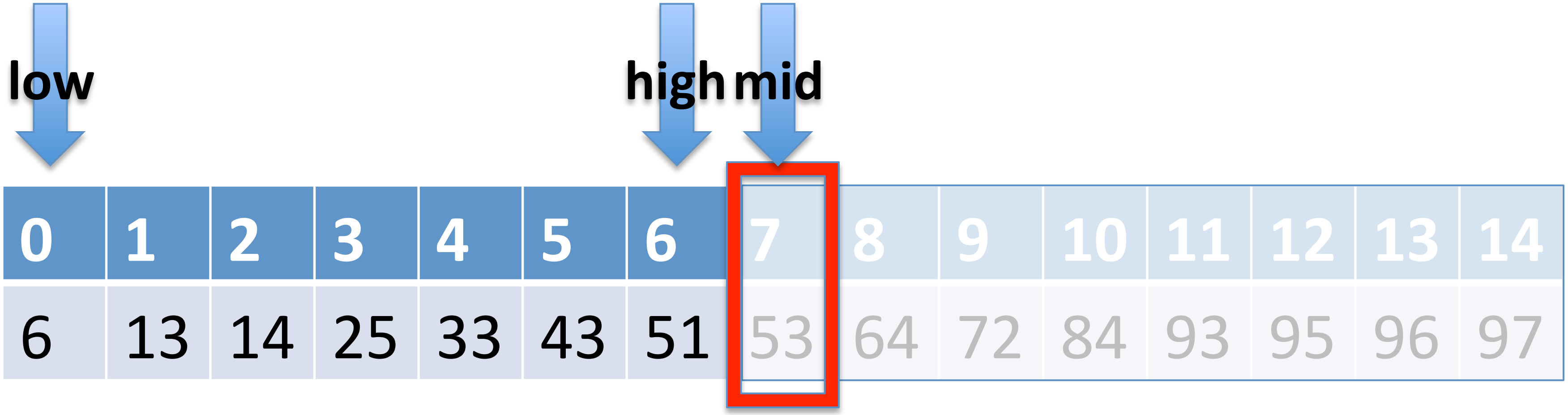
low

mid

high

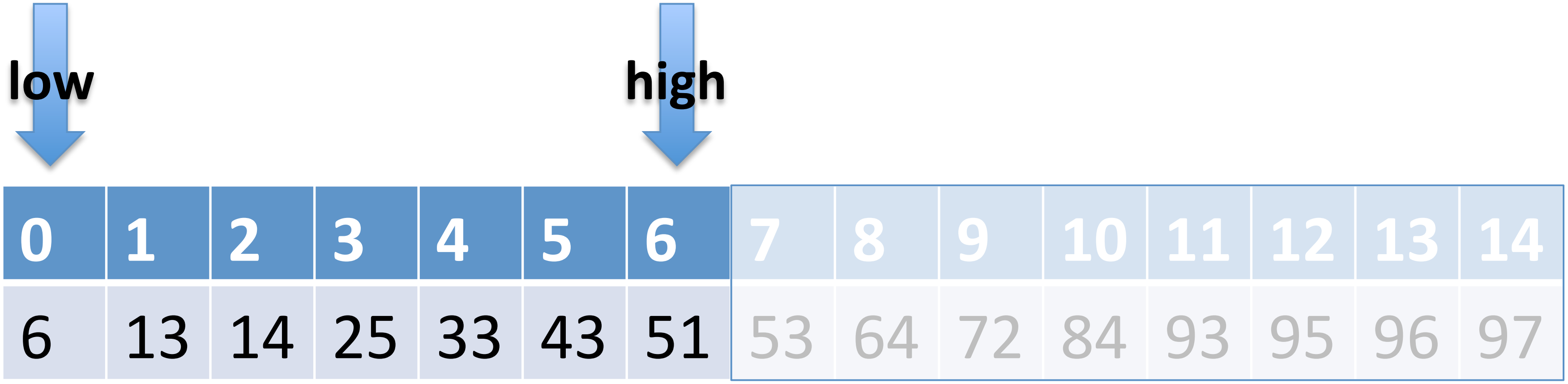
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
6	13	14	25	33	43	51	53	64	72	84	93	95	96	97

key = 33



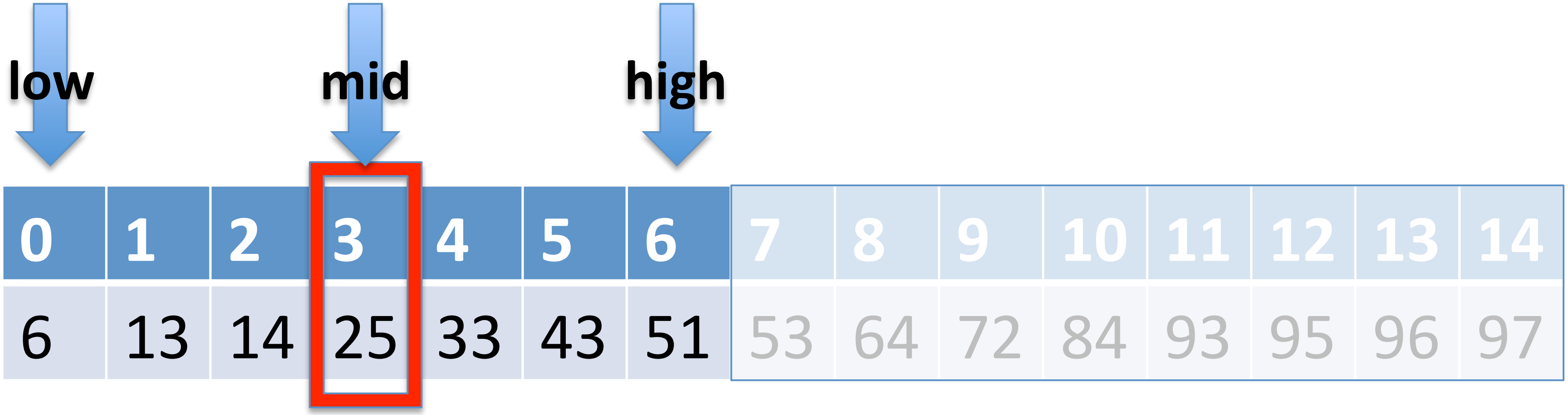
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
6	13	14	25	33	43	51	53	64	72	84	93	95	96	97

key = 33



0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
6	13	14	25	33	43	51	53	64	72	84	93	95	96	97

key = 33



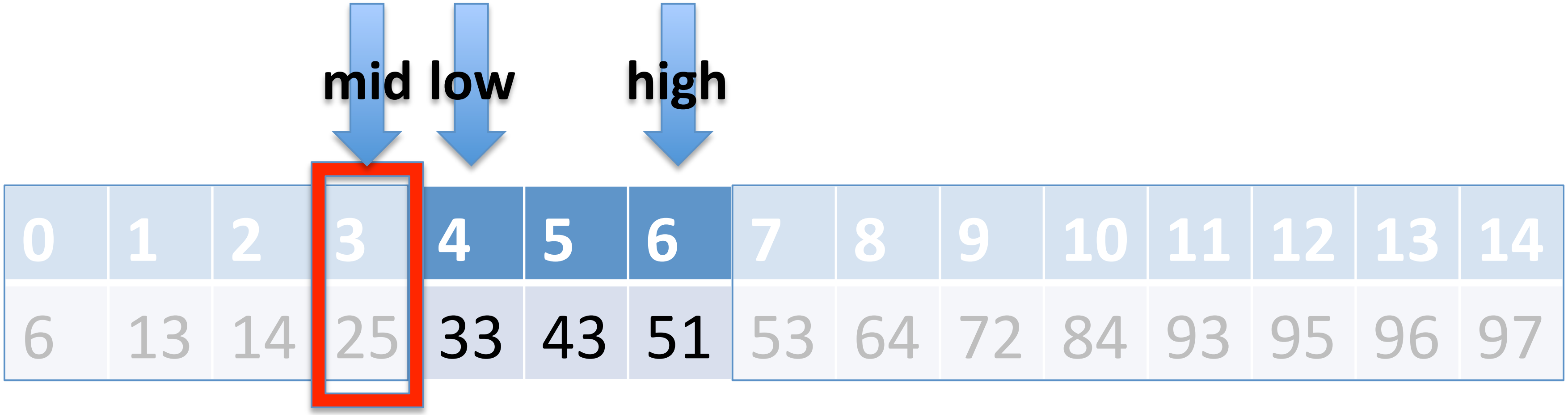
low

mid

high

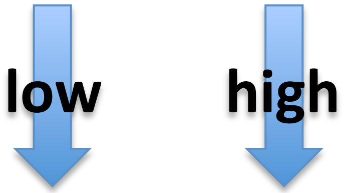
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
6	13	14	25	33	43	51	53	64	72	84	93	95	96	97

key = 33



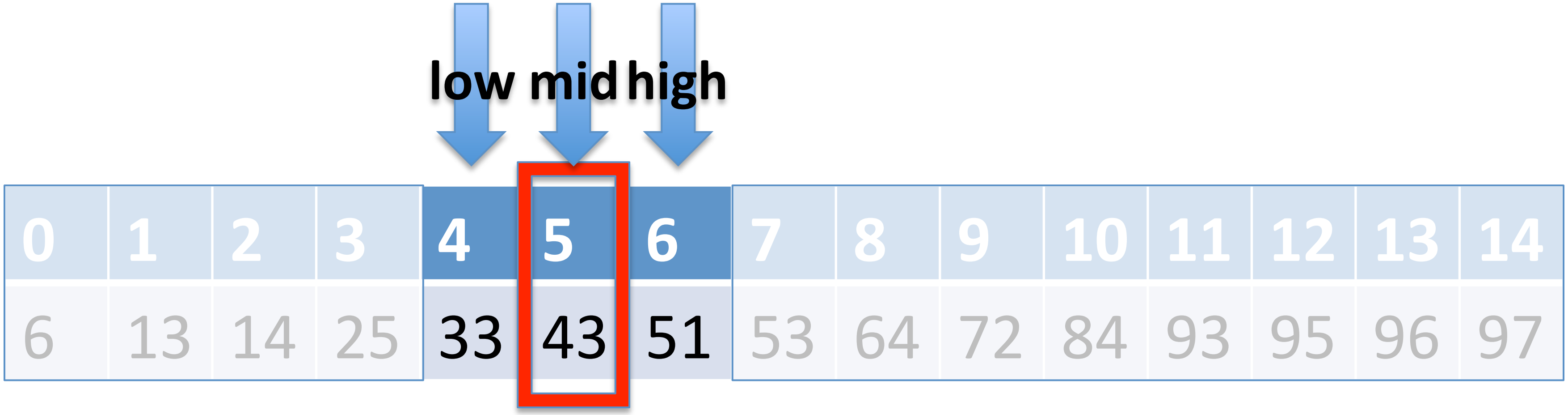
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
6	13	14	25	33	43	51	53	64	72	84	93	95	96	97

key = 33



0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
6	13	14	25	33	43	51	53	64	72	84	93	95	96	97

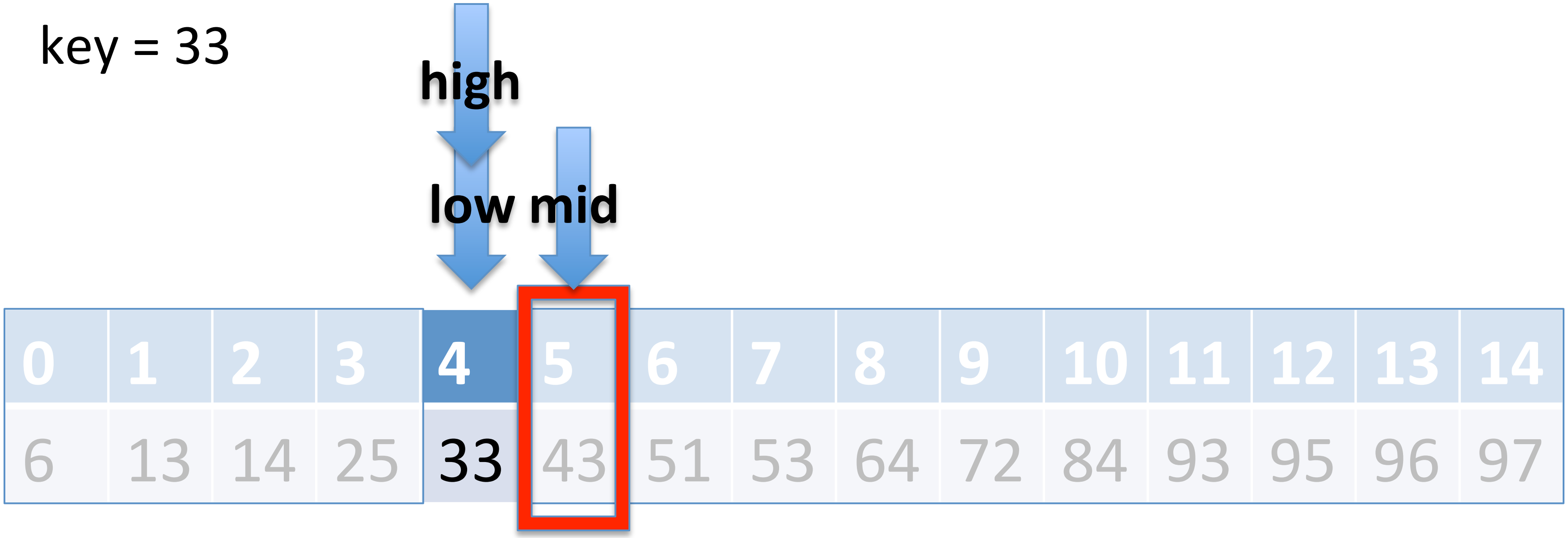
key = 33



low mid high

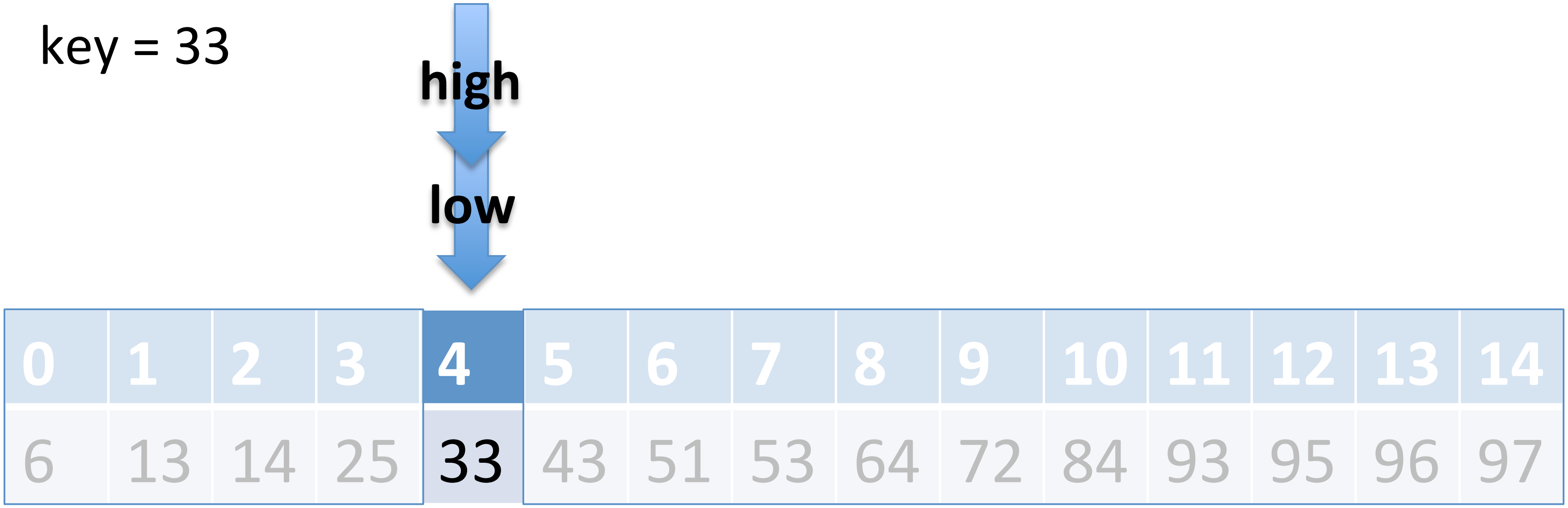
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
6	13	14	25	33	43	51	53	64	72	84	93	95	96	97

key = 33



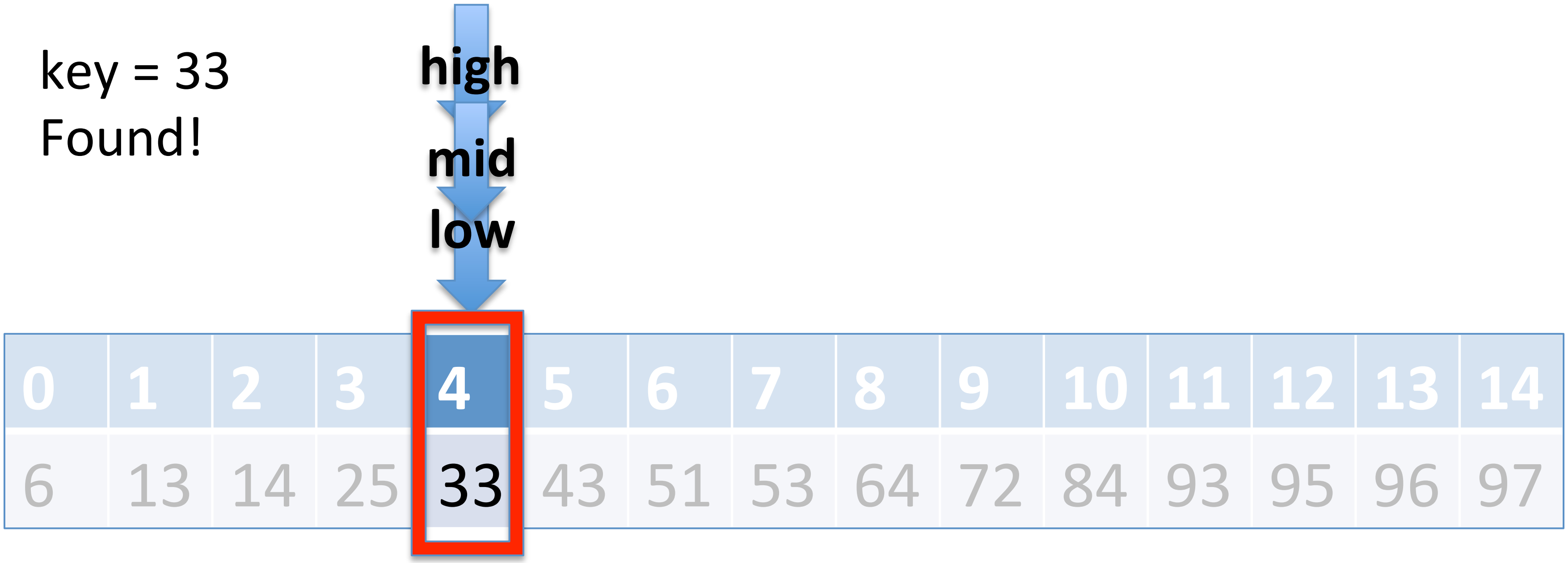
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
6	13	14	25	33	43	51	53	64	72	84	93	95	96	97

key = 33



0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
6	13	14	25	33	43	51	53	64	72	84	93	95	96	97

key = 33
Found!



0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
6	13	14	25	33	43	51	53	64	72	84	93	95	96	97

- If low > high, we know the item is not found (stop).
- If array[mid] == key, item is found (stop).
- If array[mid] > key, repeat algorithm ***with only the left half of the array***.
  - How do we change low & high?
- If array[mid] < key, repeat algorithm ***with only the right half of the array.***
  - How do we change low & high?

- If low > high, we know the item is not found (stop).
- If array[mid] == key, item is found (stop).
- If array[mid] > key, repeat algorithm ***with only the left half of the array***.
  - How do we change low & high?
  - high = mid - 1
- If array[mid] < key, repeat algorithm ***with only the right half of the array.***
  - How do we change low & high?
  - low = mid + 1

Recursive formulation

- Function: binary_search(A, key, low, high)
- Base cases:
  - Found key: Return position found.
  - low > high: Return -1 (indicating not found).
- Recursive cases:
  - array[mid] > key: binary_search(A, key, low, mid – 1)
  - array[mid] < key: binary_search(A, key, mid + 1, high)