Lecture 22:
Algorithms for Sorting and Searching

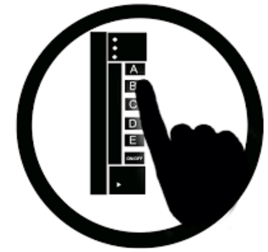
CS 1110

Introduction to Computing Using Python

[E. Andersen, A. Bracy, D. Gries, L. Lee, S. Marschner, C. Van Loan, W. White]

The first two clicker questions are special.

We will be calling one person's name out after the poll closes and you will be given a prize.



Did you have breakfast?

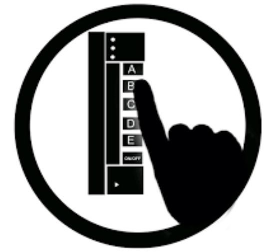
A: no, I never have breakfast

B: no, and I’m starving.

C: I had coffee. That is my breakfast.

D: I had breakfast.

E: I had a really great breakfast!



The first two clicker questions are special.

We will be calling one person's name out after the poll closes and you will be given a prize.

What is your t-shirt size?

A: Small

B: Medium

C: Large

D: X-Large

E: I do not want a free t-shirt.

Announcements

- Remember:
  - When you call an instance method,
    - call it via the object
    - (We're seeing a lot of ppl calling it via the class name) the test cases won't catch this, but this is a style/concept issue for which you will lose points:

```
c1 = Circle(1,2,3)
c1.draw()
NOT
Circle.draw(c1)
```

Algorithms for Search and Sort

- Moving beyond correctness!
- Our approach:
  - review programming constructs (`while` loop) and analysis
  - no built-in methods such as `index`, `insert`, `sort`, etc.
- Today we'll discuss
  - Linear search
  - Binary search
  - Insertion sort
- More on sorting next lecture
- More on the topic in next course, CS 2110!

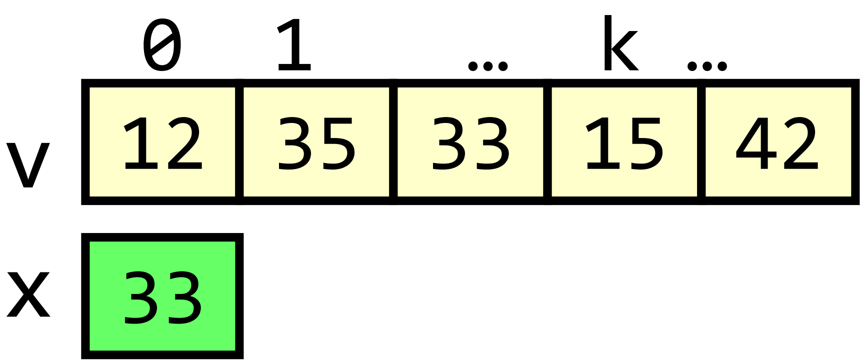
Searching for an item in a collection

Is the collection organized? What is the organizing scheme?

Indiana Jones and the Raiders of the Lost Ark

Searching in a List

- Search for x in a list v
- Start at index 0, keep checking *until* you find it



Searching in a List

- Search for x in a list v
- Start at index 0, keep checking *until* you find it or *until no more element to check*

	0	1	…	k	…
v	12	35	33	15	42
x	14				

Linear search

See `search.py`

Effort to do Linear Search (Q)

- Search for x in a list v
- Start at index 0, keep checking *until* you find it or *until no more element to check*

	0	1	…	k	…
v	12	35	33	15	42
x	14				

Linear search

Suppose list v doubles in length. The expected "effort" required to perform the linear search is

A. Squared
B. Doubled
C. A bit more
D. The same
E. I don't know

See `search.py`

Search Algorithms

- Search for x in a list v
- Start at index 0, keep checking *until* you find it or *until no more elements to check*

v	12	35	33	15	42
x	14				

Linear search

- Search for x in a *sorted* list v

Searching in a sorted list should require less work!

v	12	15	33	35	42
x	14				

Binary search

How do you search for a word in a dictionary? (NOT linear search)

To find the word “**Tierartz**” in my German dictionary…

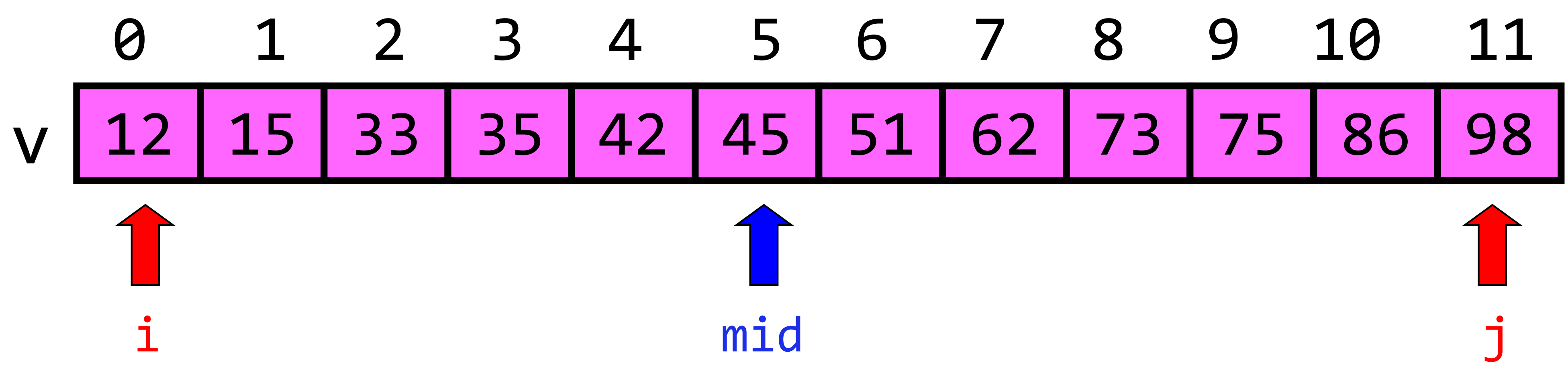
```
while dictionary is longer than 1 page:
   open to the middle page
      if last word of 1st half comes before Tierartz:
           Rip* and throw away the 1st half
     else:
           Rip* and throw away the 2nd half
```

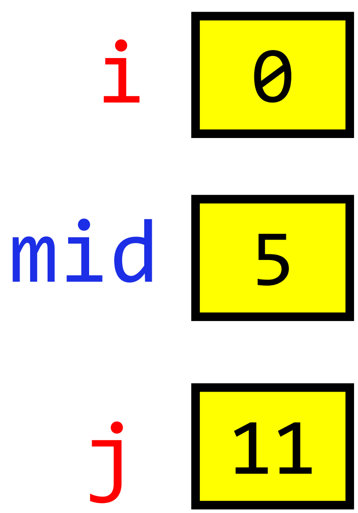
** For dramatic effect only--don’t actually rip your dictionary! Just pretend that the part is gone.*

Repeated halving of “search window”

```
Original:           3000 pages
After 1 halving:    1500 pages
After 2 halvings:    750 pages
After 3 halvings:    375 pages
After 4 halvings:    188 pages
After 5 halvings:     94 pages
      :
After 12 halvings:     1 page
```

Binary Search: target x = 70



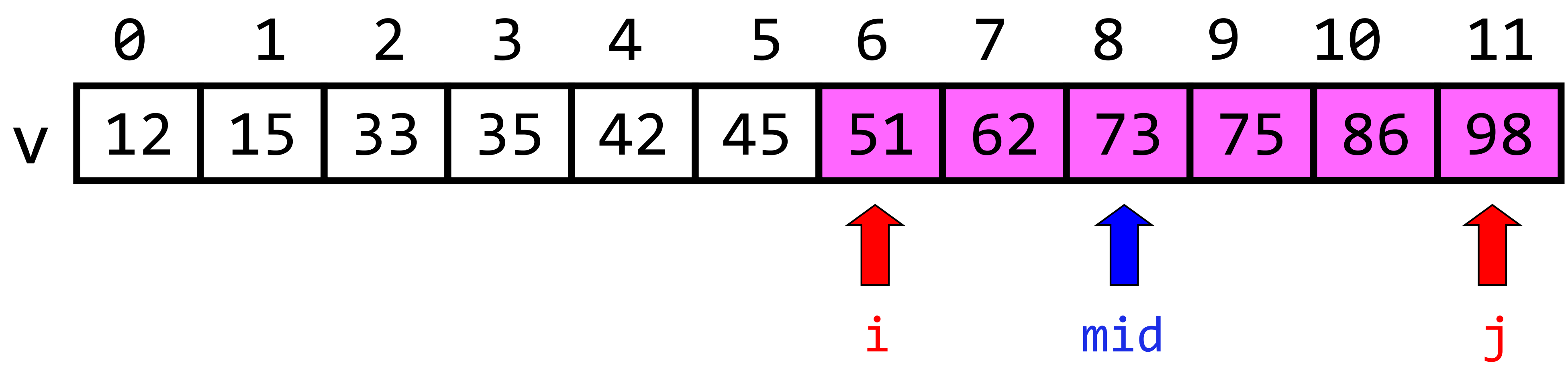


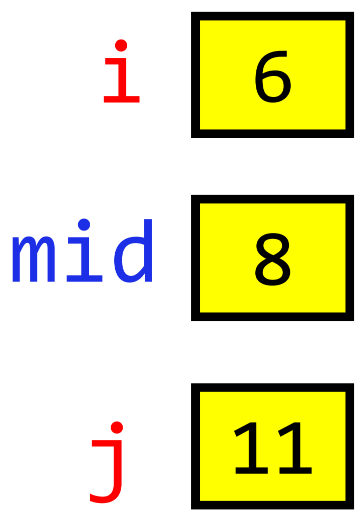
v[mid] is not x

v[mid] < x

So throw away the left half...

Binary Search: target x = 70





v[mid] is not x

x < v[mid]

So throw away the right half…

Binary Search: target x = 70

	0	1	2	3	4	5	6	7	8	9	10	11
v	12	15	33	35	42	45	51	62	73	75	86	98

mid i j

i	6
mid	6
j	7

v[mid] is not x

v[mid] < x

So throw away the left half...

Binary Search: target x = 70

	0	1	2	3	4	5	6	7	8	9	10	11
v	12	15	33	35	42	45	51	62	73	75	86	98

i mid j

i	7
mid	7
j	7

v[mid] is not x

v[mid] < x

So throw away the left half...

Binary Search: target x = 70

	0	1	2	3	4	5	6	7	8	9	10	11
v	12	15	33	35	42	45	51	62	73	75	86	98

i	8
mid	7
j	7

DONE because
i is greater than j
→ Not a valid search window

Effort to do Binary Search (Q)

- Search for x in a list v
- keep eliminating half of the list *until* you find it

  or *until no more element to check*

	i …		mid …		j
v	12	35	33	15	42
x	14				

Binary search

Suppose list v doubles in length. The expected “effort” required to perform the binary search is

A. Squared
B. Doubled
C. A bit more
D. The same
E. I don’t know

See `search.py`

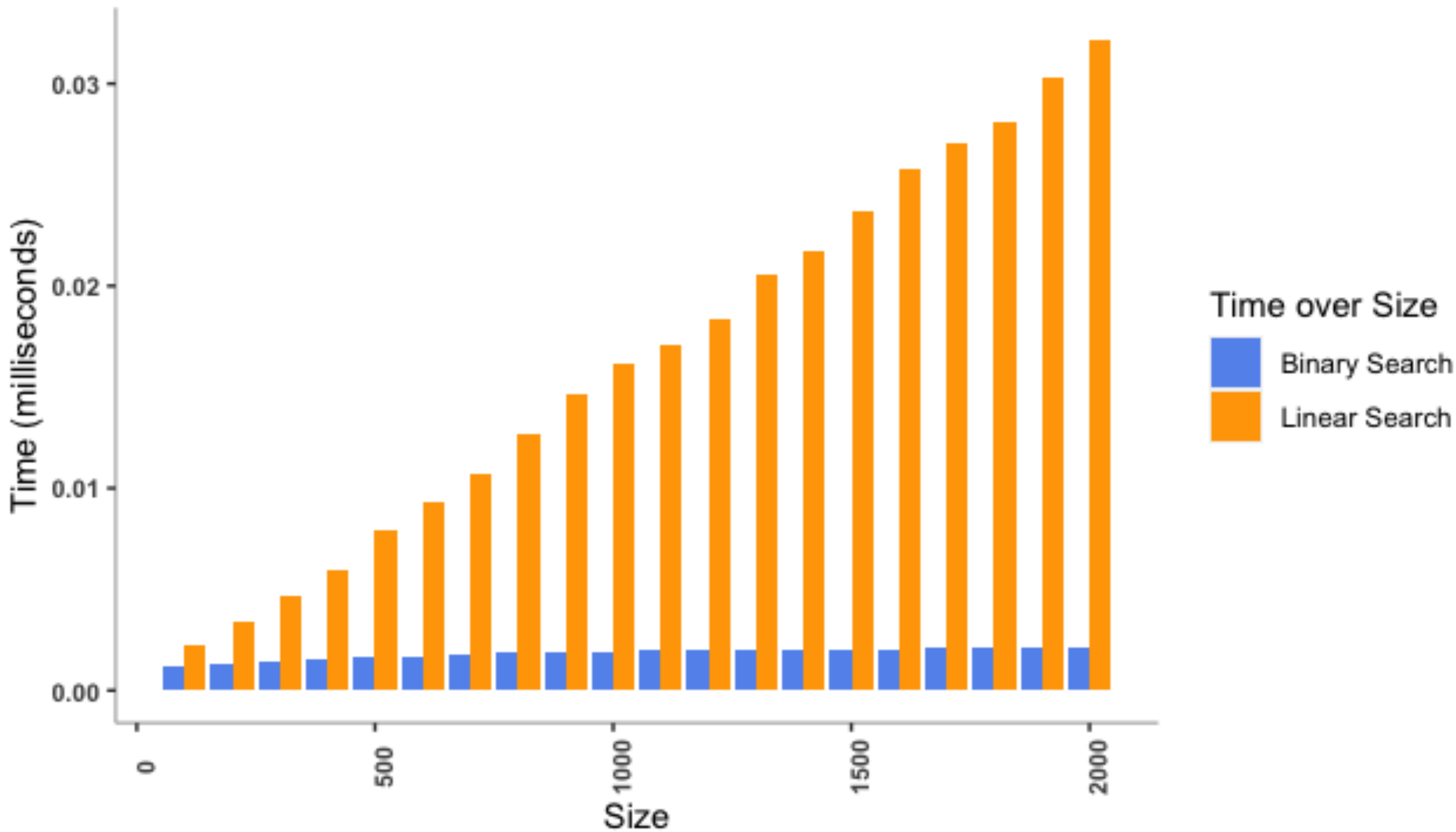
Binary Search

- Repeatedly halve the "search window"
- An item in a sorted list of length n can be located with just $\log_2 n$ comparisons.

n	log2(n)
100	7
1000	10
10000	13

- Recall: with linear search, we doubled the list and we doubled our work. With binary search, we can make the list ***100 times longer*** and not even double our work.
- "Savings" is significant!

Linear Search Versus Binary Search



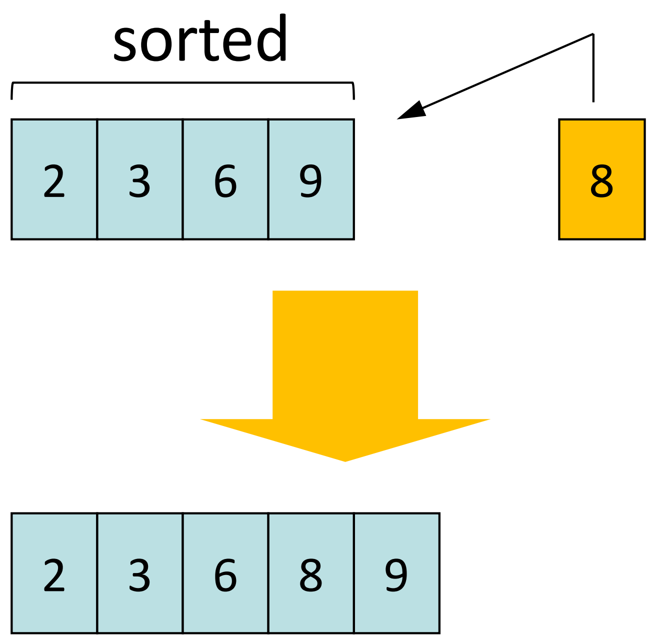
https://www.connectingdots.science/linear-vs-binary-search-visual-comparision/

Binary search is efficient, but we need to sort the vector in the first place so that we can use binary search

- Many sorting algorithms out there...
- We look at **insertion sort** now
- Next lecture we'll look at **merge sort** and do some analysis

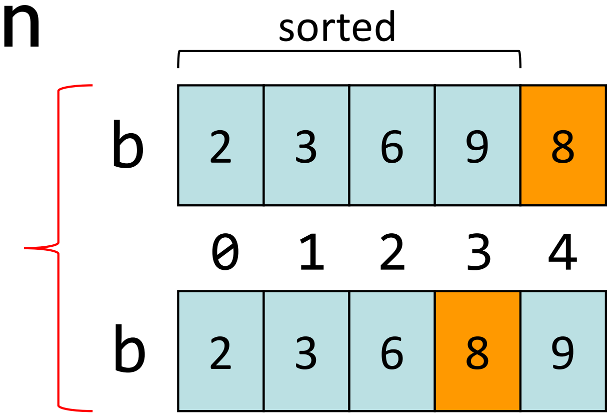
The Insertion Process

- Given a sorted list x, insert a number y such that the result is sorted
- Sorted: arranged in ascending (small to big) order



We'll call this process a "push down," as in push a value down until it is in its sorted position

Push Down

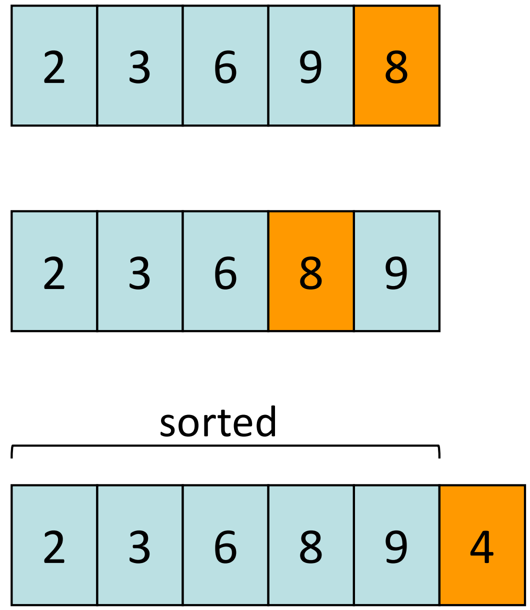


Push down 8 (b[4]) into the sorted segment b[0..3]

Just swap 8 & 9

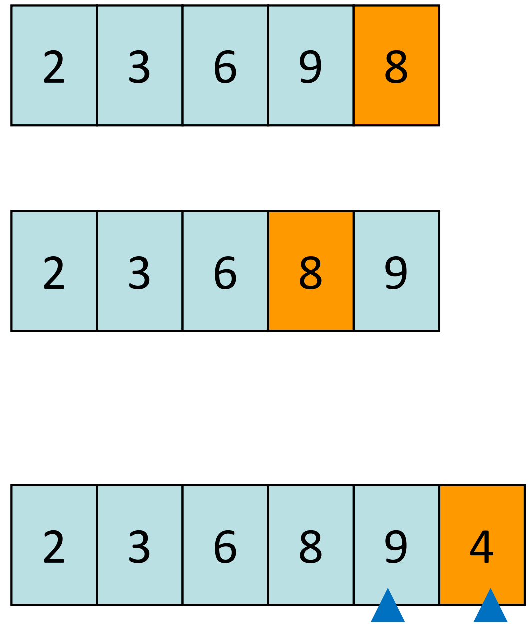
The notation **b[h..k]** means elements at indices **h** *through* **k** of list **b**, i.e., *including* **k**

Push Down



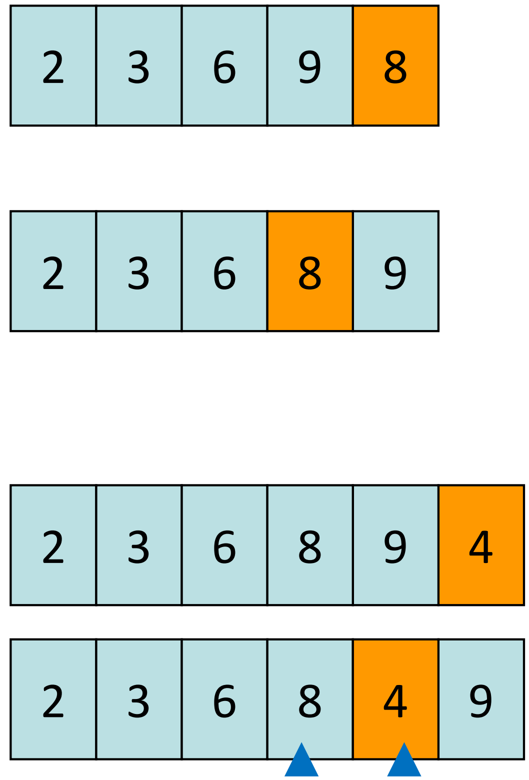
Push down 4 into the sorted segment

Push Down



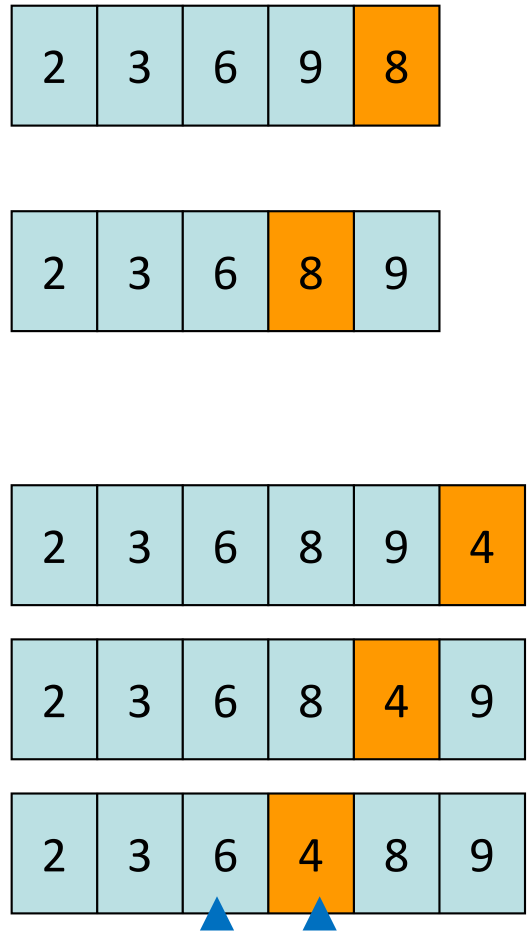
Compare adjacent components:
swap 9 & 4

Push Down



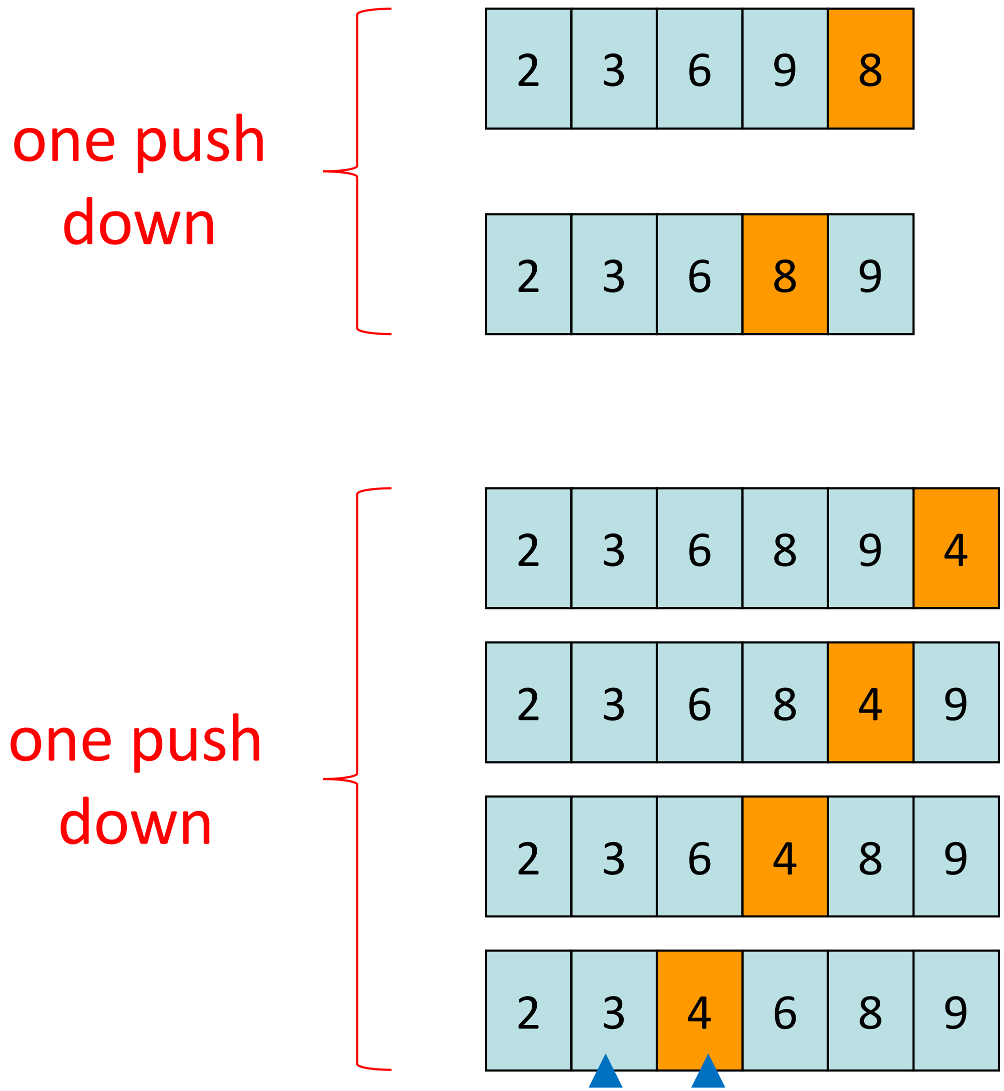
Compare adjacent components:
swap 8 & 4

Push Down



Compare adjacent components:
swap 6 & 4

Push Down

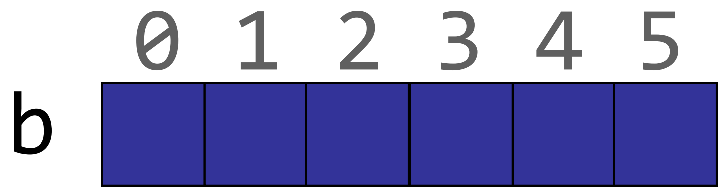


Compare adjacent components:
DONE! No more swaps.

See `push_down()` in `insertion_sort.py`

Sort list b using Insertion Sort (1)

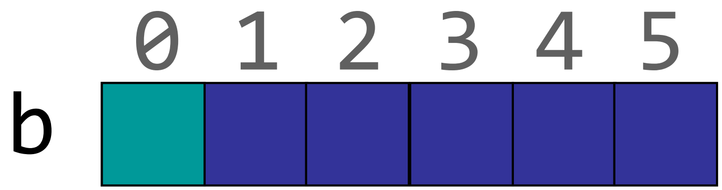
Need to start with a *sorted* segment. How do you find one?



See `insertion_sort()`

Sort list b using Insertion Sort (2)

Need to start with a *sorted* segment. How do you find one?



Length 1 segment is sorted

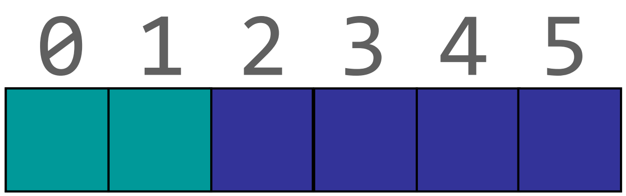
```
push_down(b, 1)
```

See `insertion_sort()`

Sort list b using Insertion Sort (3)

Need to start with a *sorted* segment. How do you find one?

b



Length 1 segment is sorted

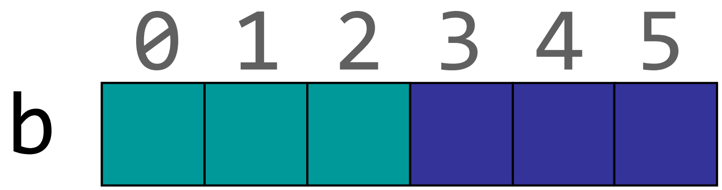
`push_down(b, 1)` Then sorted segment has length 2

`push_down(b, 2)`

See `insertion_sort()`

Sort list b using Insertion Sort (4)

Need to start with a *sorted* segment. How do you find one?



Length 1 segment is sorted

`push_down(b, 1)` Then sorted segment has length 2

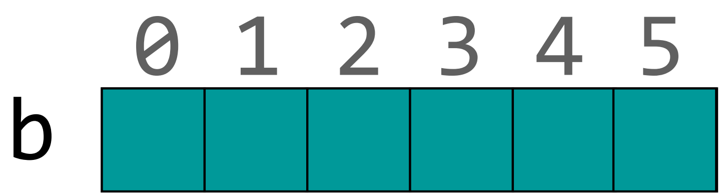
`push_down(b, 2)` Then sorted segment has length 3

`push_down(b, 3)`

See `insertion_sort()`

Sort list b using Insertion Sort (rest)

Need to start with a *sorted* segment. How do you find one?



Length 1 segment is sorted

`push_down(b, 1)` Then sorted segment has length 2

`push_down(b, 2)` Then sorted segment has length 3

`push_down(b, 3)` Then sorted segment has length 4

`push_down(b, 4)` Then sorted segment has length 5

`push_down(b, 5)` Then entire list is sorted

For a list of length n, call push_down *n-1* times.

See `insertion_sort()`

Helper functions make clear the algorithm

```
def swap(b, h, k):
    ⋮
def push_down(b, k):
  while k > 0 and b[k-1] > b[k]:
    swap(b, k-1, k)
    k= k-1

def insertion_sort(b):
    for i in range(1,len(b)):
        push_down(b, i)
```

VS.

```
def insertion_sort(b):
  for i in range(1,len(b)):
    k= i
    while (k > 0 and
        b[k-1] > b[k] ) :
          temp= b[k-1]
          b[k-1]= b[k]
          b[k]= temp
          k= k-1
```

Difficult to understand!!

Algorithm Complexity

- Count the number of comparisons needed
- In the worst case, need i comparisons to push down an element in a sorted segment with i elements.

How much work is a push down?

push down a "big" value

2	3	6	9	8

2	3	6	8	9

This push down takes 2 comparisons

push down a "small" value

2	3	6	9	1

2	3	6	1	9

2	3	1	6	9

2	1	3	6	9

1	2	3	6	9

This push down takes 4 comparisons. Worst case scenario: ***n*** comparisons needed to push down into a length ***n*** sorted segment.

Algorithm Complexity (A)

Count (approximately) the number of comparisons needed to sort a list of length n

```
def swap(b, h, k):
   ⋮

def push_down(b, k):
  while k > 0 and b[k-1] > b[k]:
        swap(b, k-1, k)
        k= k-1

def insertion_sort(b):
    for i in range(1,len(b)):
        push_down(b, i)
```

A. ~ 1 comparison
B. ~ n comparisons
C. ~ n^2 comparisons
D. ~ n^3 comparisons
E. I don't know

Algorithm Complexity Explained

- Count the number of comparisons needed
- In the worst case, need i comparisons to push down an element in a sorted segment with i elements.
- For a list of length n
  - 1st push down: 1 comparison
  - 2nd push down: 2 comparisons (worst case)

    ⋮

  - $1+2+\ldots+(n-1) = n*(n-1)/2$, say, $\mathbf{n^2}$ for big n

- For fun, check out this visualization: https://www.youtube.com/watch?v=xxcpvCGrCBc

Complexity of algorithms discussed

- Linear search: on the order of n
- Binary search: on the order of $\log_2 n$
  - Binary search is faster but requires sorted data
- Insertion sort: on the order of n^2