http://www.cs.cornell.edu/courses/cs1110/2021sp

Lecture 25:
Algorithms for Sorting and Searching

CS 1110

Introduction to Computing Using Python

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

Announcements

- Labs 17 & 18 due Friday & Monday, respectively
- Next week's discussion sections → office hours for A6 and Prelim 2
- Final Exam on May 21st 1:30-4pm. Your assigned exam session (in-person or online) will be given in CMS tomorrow. *Submit a "**regrade request**" in CMS by May 12* if you have a legitimate reason for requesting a change

Algorithms for Search and Sort

- Well known algorithms
  - focus on reviewing programming constructs (`while` loop) and analysis
  - will not use 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 in a List (Q)

- Search for a target **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 another list is twice as long as v. The expected “effort” required to do a linear search is

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

See **search.py**

Search Algorithms

- Search for a target 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 a target 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 “**tanto**” in my Spanish dictionary…

while dictionary is longer than 1 page:

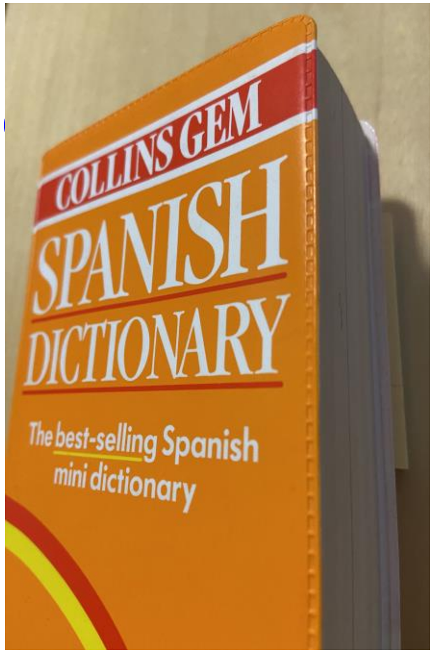
    Open to the middle page

    if first entry comes before “**tanto**”:

        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

- Repeatedly halve the “search window”
- An item in a sorted list of length n can be located with just $\log_2 n$ comparisons.
- “Savings” is significant!

n	log2(n)
100	7
1000	10
10000	13

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: 0

mid: 5

j: 11

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: 6

mid: 8

j: 11

`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

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: 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

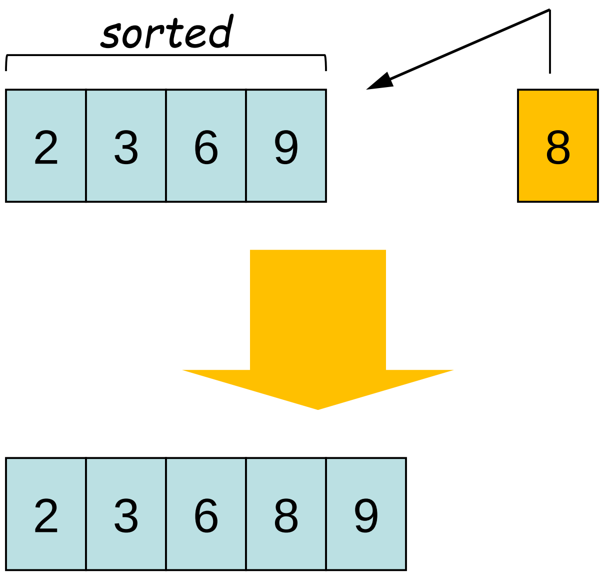
DONE because
i is greater than j
→ Not a valid search window

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

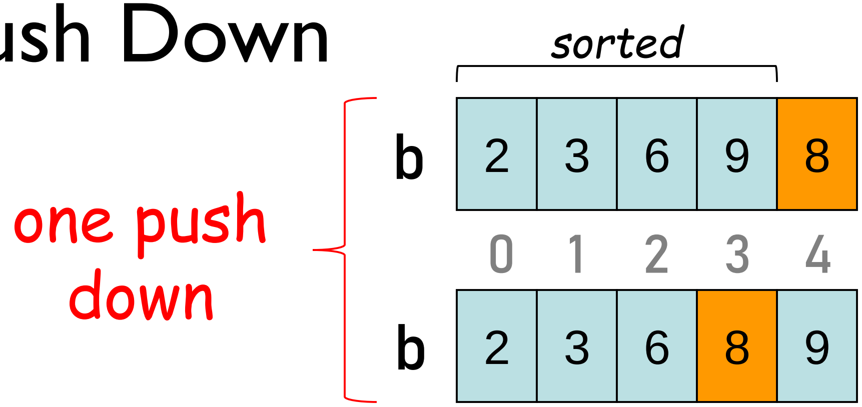
The Insertion Process

- 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]

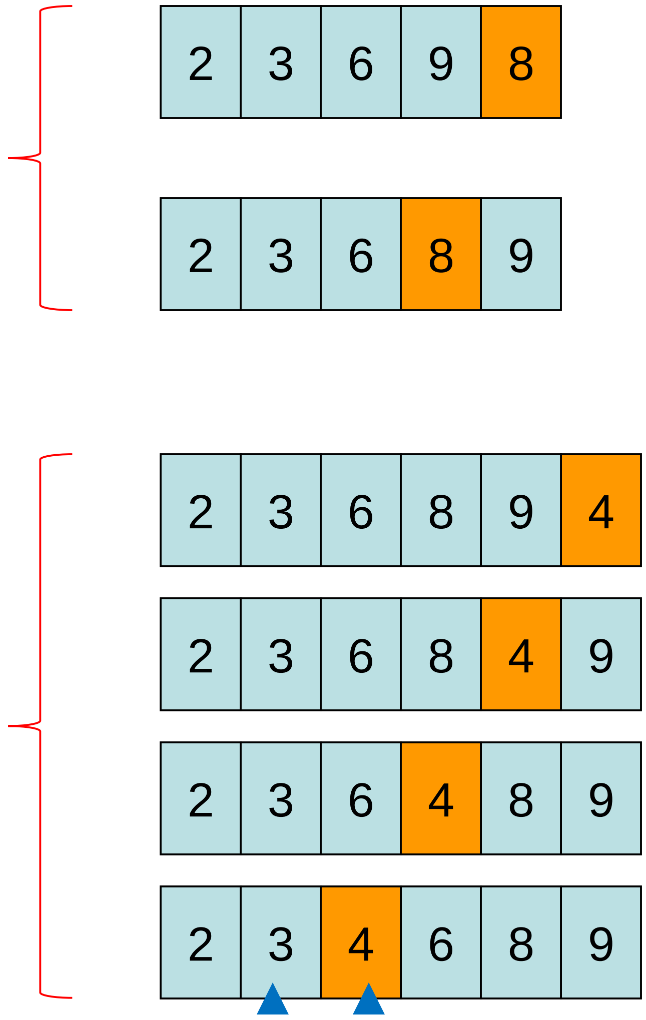
Just swap 8 & 9

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

Push Down

one push down

one push down



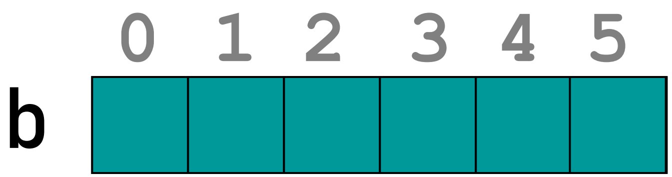
Push down 4 into the sorted segment

Compare adjacent components: swap 9 & 4
Compare adjacent components: swap 8 & 4
Compare adjacent components: swap 6 & 4
Compare adjacent components: DONE! No more swaps.

See push_down() in insertion_sort.py

Sort list b using Insertion Sort

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.

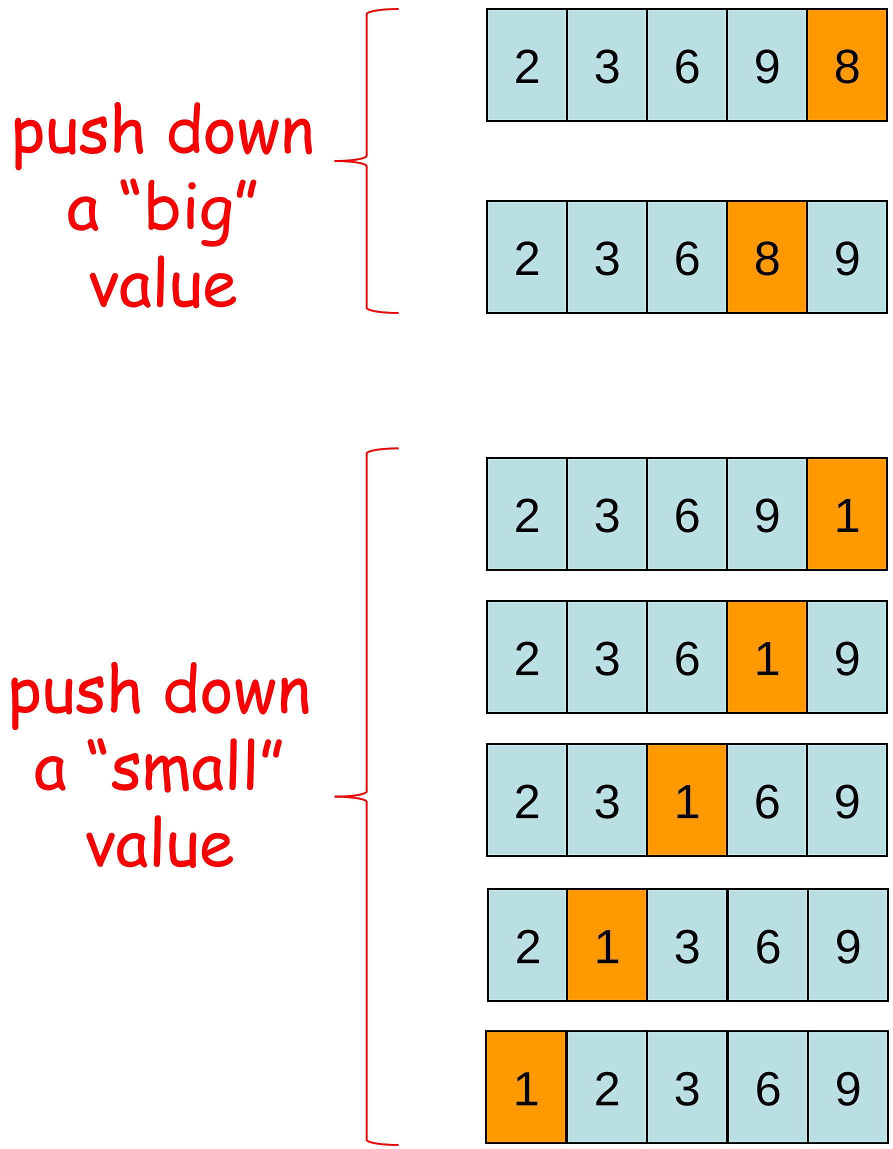
Difficult to understand!!

```
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
```

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?



This push down takes 2 comparisons

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

Algorithm Complexity (Q)

```
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)
```

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

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