CS1800 Day 20

Admin:

- hw8 (seq & series, function growth) due next Tuesday
  - notice:funny tuesday schedule given end of semester
- hw9 (algorithms)
  - due next Tuesday Dec 3 (exam3 date)
  - slightly shorter than most, points reflect this
- "exam3"
  - written to take 30 mins but you'll get 50 minutes to complete it
    - 2 math problems, 1 quick theory-ish problem
  - format identical to other exams
  - covers class 18, 19, 20
  - class 21, reccurence relations, will not be tested on exam3, is included on hw9

Content (algorithms):

- search algorithms (unordered linear search & binary search)
- sort algorithms (insertion & merge)
- quantifying (estimating) algorithm run time

TRACE (Northeastern's survey of course quality)

TRACE feedback helps me be a better teacher (in a future semester)
TRACE feedback helps NU identify strong / weak teachers

- feedback is anonymous
    - we won't get feedback until after you've received your grade
- please review both CS1800 and CS1802
    - CS1802 for recitation hour, materials, recitation related admin
    - CS1800 for everything else (lesson, homework, exam, office hours, tutorial, all other admin...)

Please take a few minutes to give feedback about what worked and what didn't in the course.
(accessible via myNortheastern or email, should have been sent out this morning)

Thank you for your earnest feedback here :)

Review: Log Operation

$$2^3 = 8 \longleftrightarrow \mathrm{LOG}_2 8 = 3$$

$\mathrm{LOG}_B X$ IS THE POWER OF B EQUAL TO X

$$\mathrm{LOG}_3 27 = 3$$

HOW MANY TIMES DO I NEED TO MULTIPLY BY 3 TO GO FROM 1 TO 27?

In Class Activity (log practice)

Solve for x in each of the equalities below

log_10 1000 = x $= 3$

$10^3 = 1000$

log_2 16 = x $= 4$

$2^4 = 16$

log_2 x = 10

$2^{10} = 1024$

$x = 1024$

log_x 125 = 3

$x^3 = 125$

$x = 5$

log_2 16 + log_2 32 = x

$4 + 5 = x$

✓ ✓

log_2 (16 * 32) = x

$16 \cdot 32 = 2^4 \cdot 2^5 = 2^9$

$\rightarrow \mathrm{LOG}_2\, 16 \cdot 32 = \mathrm{LOG}_2\, 2^9 = 9$

(++) write a general rule for the sum of logs with the same base which this example suggests

List Convention: Let's start indexing our lists at zero

L =

14	102	-4	6
L[0]	L[1]	L[2]	L[3]

Definitions:

"Search": Find index of first occurance of an item in a list

0 1 2 3 4 5 6 7

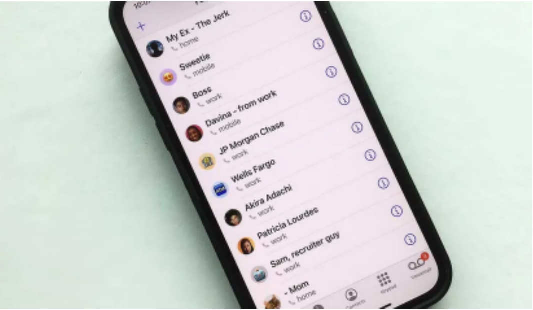
Given the following list: [2, -2, 100, 2.347, 4, 100, 5, -17]

- search question: find the index of 2	- search output: 0 is index of first 2
- search question: find the index of 100	- search output: 2 is index of first 100
- search question: find the index of 18	- search output: 18 isn't in the list

"Sort": given a list of items, order them from least to greatest (equal items in any order)

Sort input: [6, 3, 2, 100, -5, 3]	Sort output: [-5, 2, 3, 3, 6, 100]

Why search?



Why sort?

- sorted lists are quicker to operate on (see binary search vs unordered linear search)

- sorted list positions offer insights
    - first item is minimum
    - last item is maximum
    - item in middle is median
    - if "bob" isn't between "alice" and "chuck" in a sorted list, then bob not in list

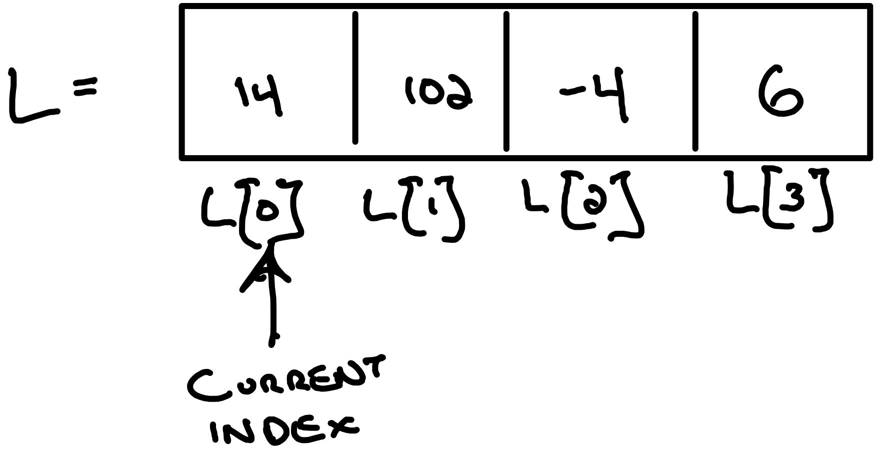
Search: Unordered Linear Search

search inputs: a list and an item to search for

Intuition: Starting at first index in list, check if equal to item, move rightward until item found

FIND 6 IN LIST BELOW

Example:



$L[0] \neq 6$ so

WE CHECK NEXT INDEX

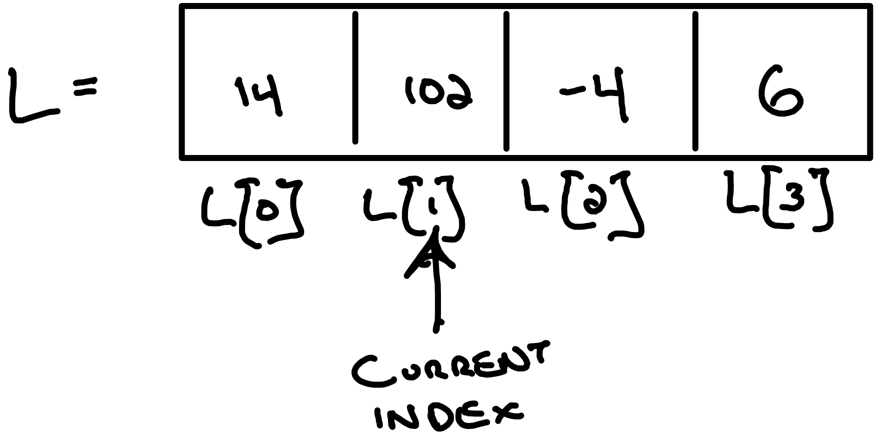
Search: Unordered Linear Search

search inputs: a list and an item to search for

Intuition: Starting at first index in list, check if equal to item, move rightward until item found

FIND 6 IN LIST BELOW

Example:



$L[1] \neq 6$ SO WE CHECK NEXT INDEX

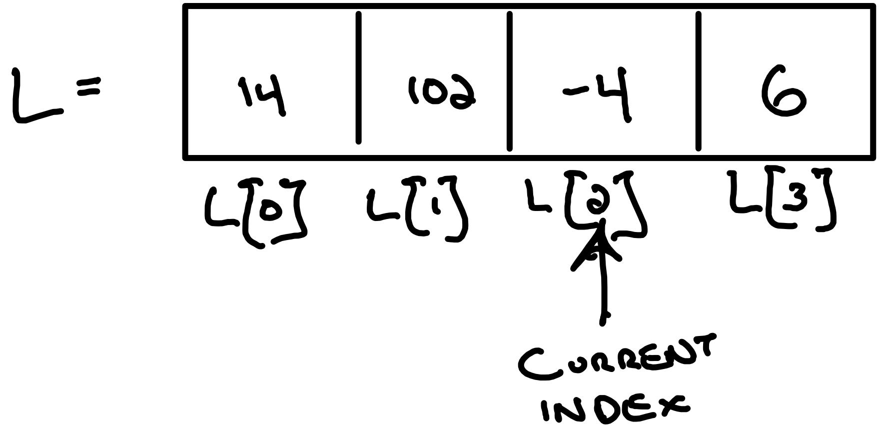
Search: Unordered Linear Search

search inputs: a list and an item to search for

Intuition: Starting at first index in list, check if equal to item, move rightward until item found

FIND 6 IN LIST BELOW

Example:



$L[2] \neq 6$ SO WE CHECK NEXT INDEX

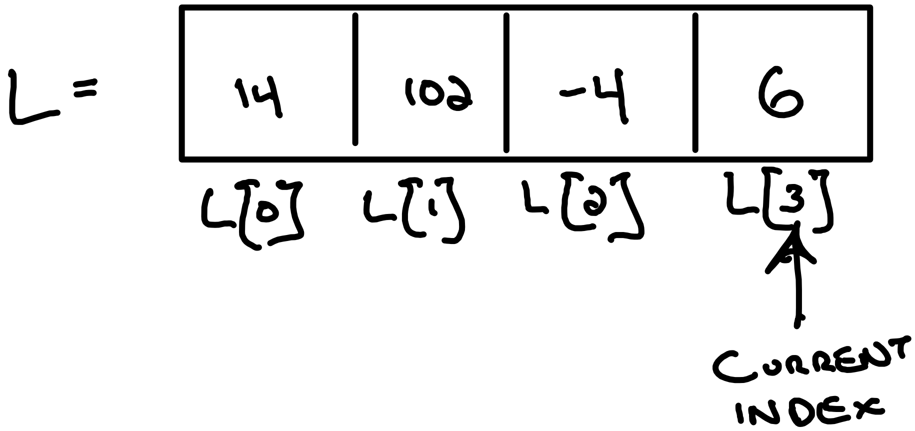
Search: Unordered Linear Search

search inputs: a list and an item to search for

Intuition: Starting at first index in list, check if equal to item, move rightward until item found

FIND 6 IN LIST BELOW

Example:



L[3] = 6 SO WE RETURN 3

Is this algorithm any good? What do we want from our algorithms?

- Correctness

- Low memory use: doesn't require the computer to store too much data at any moment

- Quick runtimes: completes the task in as few "operations" as possible for input of size n

- Simplicity: we humans have to build and maintain this thing. simplicity reduces the chance that we'll make an error

In practice (and in CS1800) folks usually focus on the runtimes of correct algorithms.

Quantifying runtime:

Runtime: how many "operations" required to complete algorithm for input of size n

To simplify our analysis of algorithms:

- lets only count comparisons (is item0 less than, equal to, or greater than item1?)

<whole class card demo: counting operations in a few unordered linear searches>

(punchline: different inputs require different number of comparisons)

Quantifying runtime:

Runtime: how many "operations" required to complete algorithm for input of size n

To simplify our analysis of algorithms:

- lets only count comparisons (is item0 less than, equal to, or greater than item1?)
- lets assume the worst possible input for a given algorithm (requiring the most comparisons)

In the worst case, for an input list with n items:

- unordered linear search requires we compare our item to every input: $T(n) = n$

OF COMPARISONS

<show binary search with cards>

Search: Binary Search

⚠ BINARY SEARCH ASSUMES SORTED LIST

search inputs: a sorted list and an item to search for

Intuition: compare item to mid-point part of list which might contain item, update & repeat as needed

FIND INDEX OF 11 IN LIST BELOW

Example:

1	4	11	17	21	27	30
L[0]	L[1]	L[2]	L[3]	L[4]	L[5]	L[6]

Search: Binary Search

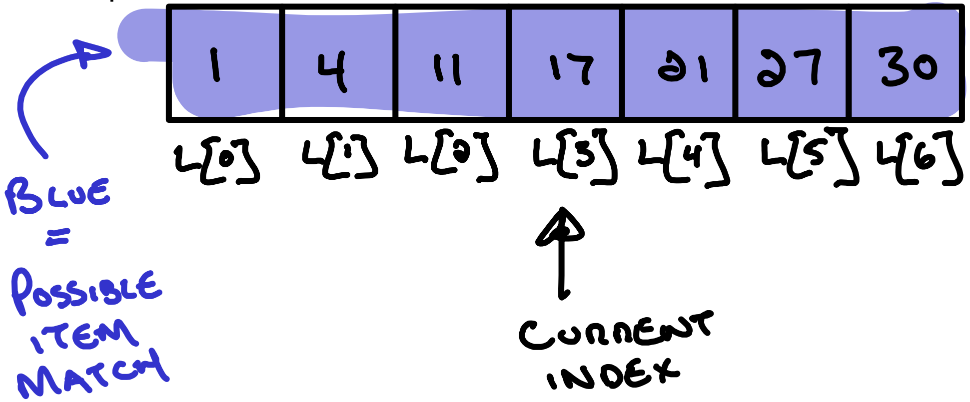
⚠ Binary Search assumes sorted list

search inputs: a sorted list and an item to search for

Intuition: compare item to mid-point part of list which might contain item, update & repeat as needed

Find index of 11 in list below

Example:



$11 < L[3] = 17$

So we restrict to smaller index than 3

Search: Binary Search

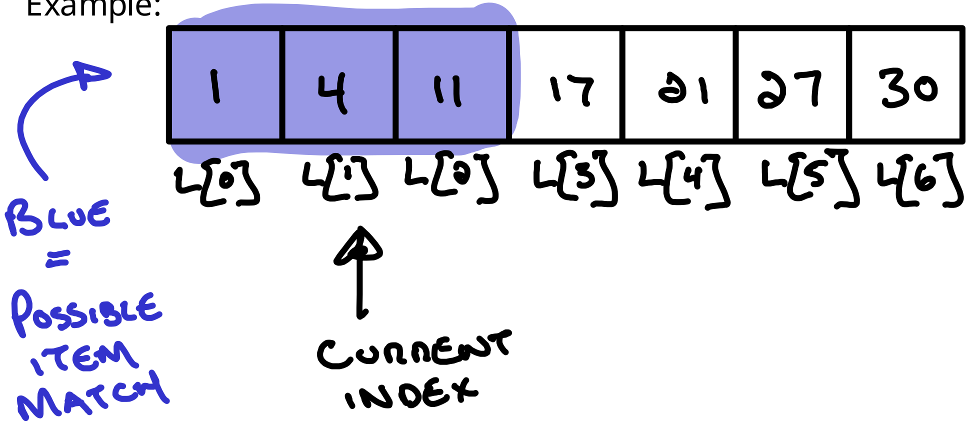
⚠ BINARY SEARCH ASSUMES SORTED LIST

search inputs: a sorted list and an item to search for

Intuition: compare item to mid-point part of list which might contain item, update & repeat as needed

FIND INDEX OF 11 IN LIST BELOW

Example:



1	4	11	17	21	27	30
L[0]	L[1]	L[2]	L[3]	L[4]	L[5]	L[6]

$11 > L[1] = 4$

SO WE RESTRICT TO LARGER INDEX THAN 1

Search: Binary Search

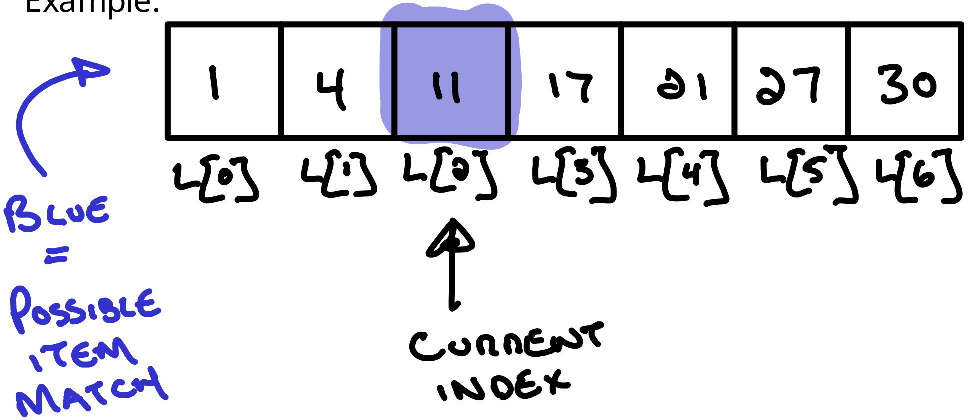
⚠ BINARY SEARCH ASSUMES SORTED LIST

search inputs: a sorted list and an item to search for

Intuition: compare item to mid-point part of list which might contain item, update & repeat as needed

FIND INDEX OF 11 IN LIST BELOW

Example:



11 = L[2] = 11

WE FOUND AN 11 AT INDEX 2

In Class Activity:

2,3,4,5,6,7,8

- Build an example (target item & list of size 7) where binary search works quickest (fewest comparison)

searching for an item which is located in the middle of the list takes only 1 comparison

- Build an example (target item & list of size 7) where binary search works slowest (most comparisons)

searching for item which isn't in the list ensures we don't stop early, this is the worst case
(in this example of list of size 7, it uses 3 comparisons)

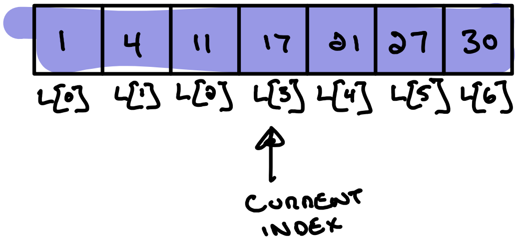
- For a list of size n, what is the most comparisons binary search will require to complete?
  (hint: coming up with an exact expression can be tough here, feel free to approximate as needed to keep it simple. It can feel funny to approximate like this at first, but we'll justify it with our Big-O definition of function growth)

Worst Case Performance of Binary Search

Notice:

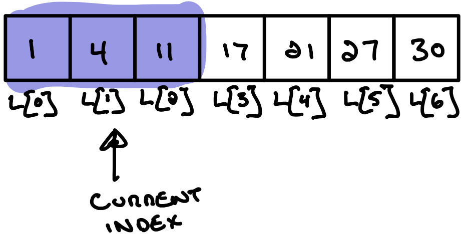
- the "worst case" of binary search is when we cannot stop early for having found target item
- Each comparison cuts the set of possible matching indexes (blue shaded area) in *half

Previous Example (target item is 11):



$11 < L[3]=17$

SO WE RESTRICT
TO SMALLER INDEX
THAN 3



Clearly, with 1 comparison we can run binary search on a list of size n=1. So...

- 2 comparisons run binary search (worst case) on a list of size n=2
- 3 comparisons run binary search (worst case) on a list of size n=4
- 4 comparisons run binary search (worst case) on a list of size n=8
- n comparisons run binary search (worst case) on a list of size 2^{n-1}

Remember logs?

$$2^3 = 8 \longleftrightarrow \log_2 8 = 3$$

$\log_B X$ IS THE POWER OF B EQUAL TO X

So how many comparisons, does binary search use on a list of size n, in the worst case?

N COMPARISONS FOR LIST OF SIZE 2^N $\longleftrightarrow$ $\log_2 N$ COMPARISONS FOR LIST OF SIZE N

Quantifying runtime:

Runtime: how many "operations" required to complete algorithm for input of size n

To simplify our analysis of algorithms:

- lets only count comparisons (is item0 less than, equal to, or greater than item1?)
- lets assume the worst possible input for a given algorithm (requiring the most comparisons)

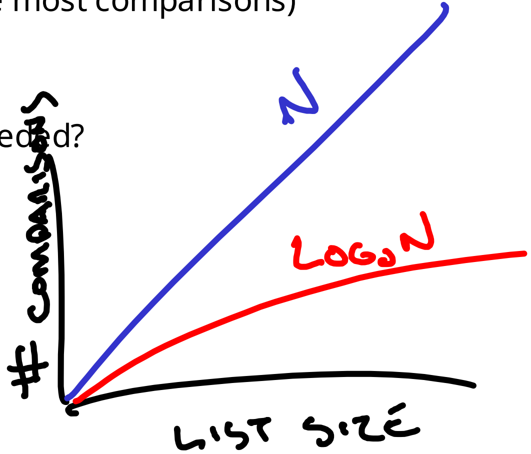
In the worst case, for an input list with n items how many comparisons are needed?

- unordered linear search

$$T_{LINEAR}(N) = N$$

- binary search

$$T_{BINARY}(N) = LOG_2 N$$



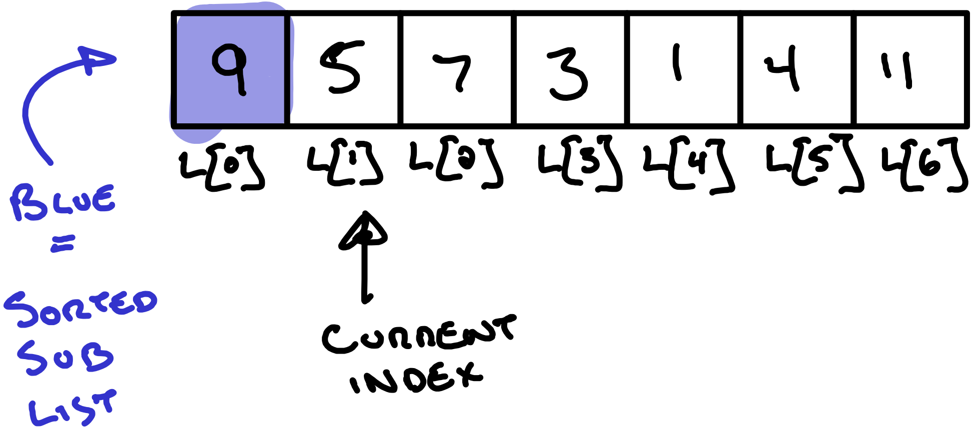
<insertion sort with cards>

Sort: Insertion Sort

sort inputs: a list

Intuition: add items, one-by-one, into a sorted sub-list (the first items in the list)

Example:



BLUE = SORTED SUB LIST

CURRENT INDEX

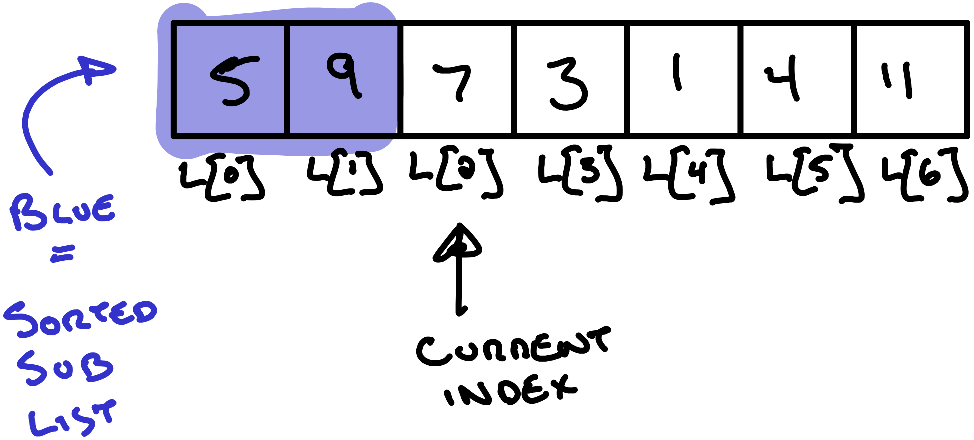
BECAUSE ANY SINGLE ITEM IS A SORTED SUB-LIST OF LENGTH 1 START INDEX AT 2ND ITEM

Sort: Insertion Sort

sort inputs: a list

Intuition: add items, one-by-one, into a sorted sub-list (the first items in the list)

Example:



5	9	7	3	1	4	11
L[0]	L[1]	L[2]	L[3]	L[4]	L[5]	L[6]

BLUE = SORTED SUB LIST

CURRENT INDEX

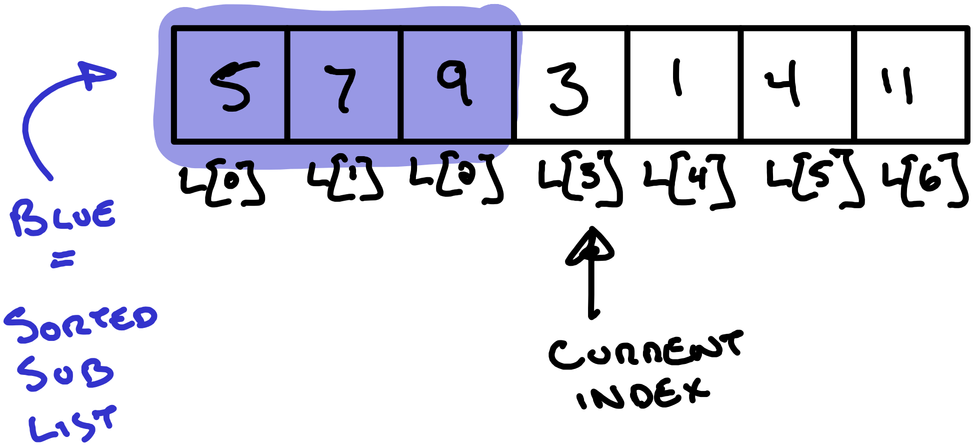
INSERTED 5 INTO SORTED SUB-LIST

Sort: Insertion Sort

sort inputs: a list

Intuition: add items, one-by-one, into a sorted sub-list (the first items in the list)

Example:



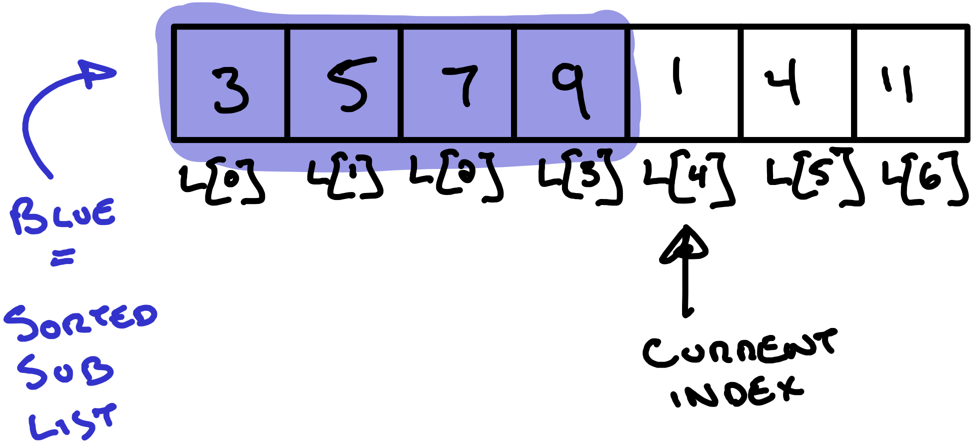
INSERTED 7 INTO SORTED SUB-LIST

Sort: Insertion Sort

sort inputs: a list

Intuition: add items, one-by-one, into a sorted sub-list (the first items in the list)

Example:



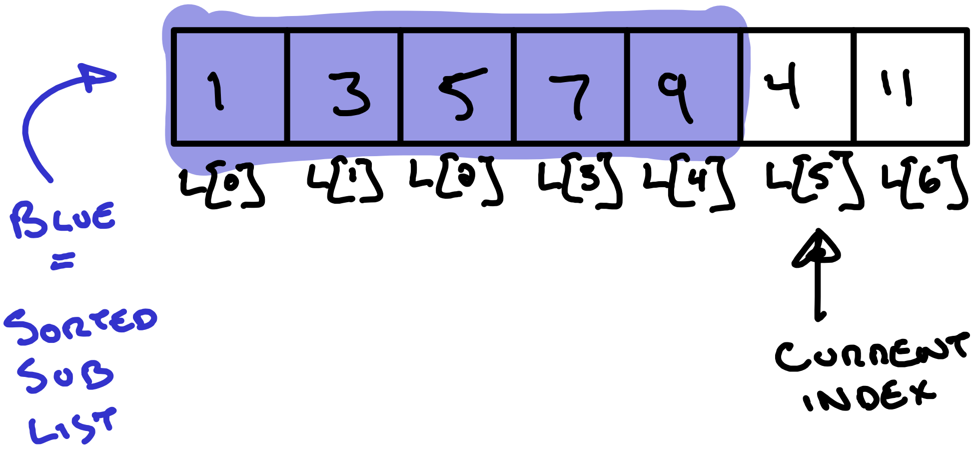
INSERTED 3 INTO SORTED SUB-LIST

Sort: Insertion Sort

sort inputs: a list

Intuition: add items, one-by-one, into a sorted sub-list (the first items in the list)

Example:



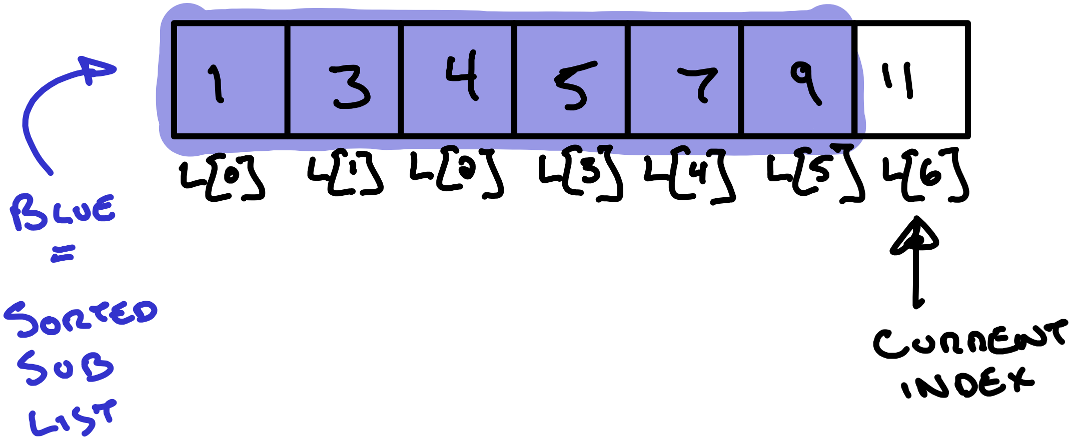
INSERTED 1 INTO SORTED SUB-LIST

Sort: Insertion Sort

sort inputs: a list

Intuition: add items, one-by-one, into a sorted sub-list (the first items in the list)

Example:



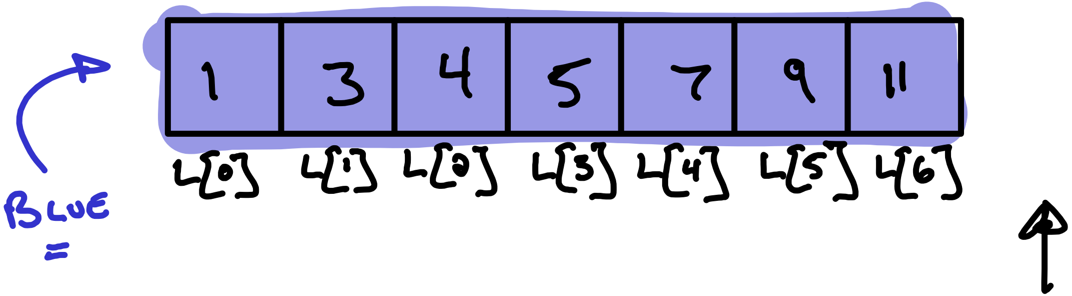
INSERTED 4 INTO SORTED SUB-LIST

Sort: Insertion Sort

sort inputs: a list

Intuition: add items, one-by-one, into a sorted sub-list (the first items in the list)

Example:



1	3	4	5	7	9	11
L[0]	L[1]	L[2]	L[3]	L[4]	L[5]	L[6]

BLUE = SORTED SUB LIST

INSERTED 11 INTO SORTED SUB-LIST

CURRENT INDEX (DONE)

SHOWING INSERTION SORT ON HW9

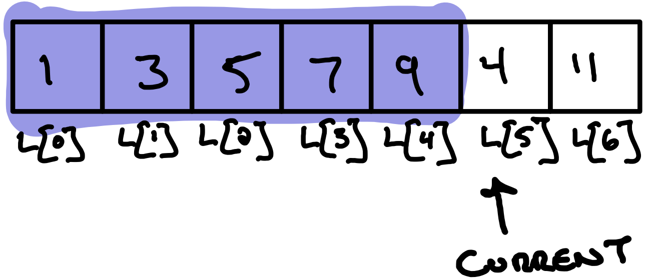
Phase	Processed				◇	Unprocessed			
0	◇	34	16	12	11	54	10	65	37
1	34	◇	16	12	11	54	10	65	37
2	16	34	◇	12	11	54	10	65	37
3	12	16	34	◇	11	54	10	65	37
4	11	12	16	34	◇	54	10	65	37
5	11	12	16	34	54	◇	10	65	37
6	10	11	12	16	34	54	◇	65	37
7	10	11	12	16	34	54	65	◇	37
8	10	11	12	16	34	37	54	65	◇

EVERYTHING LEFT OF SYMBOL IS SORTED

In Class Activity

Build an input list of length 5 which requires as many (and as few) comparisons as possible for insertion sort to complete.

ASSUME INSERTION SORT STARTS COMPARISONS FROM LEFT



COMPARISON 1: 4 > 1

COMPARISON 2: 4 > 3

COMPARISON 3: 4 < 5

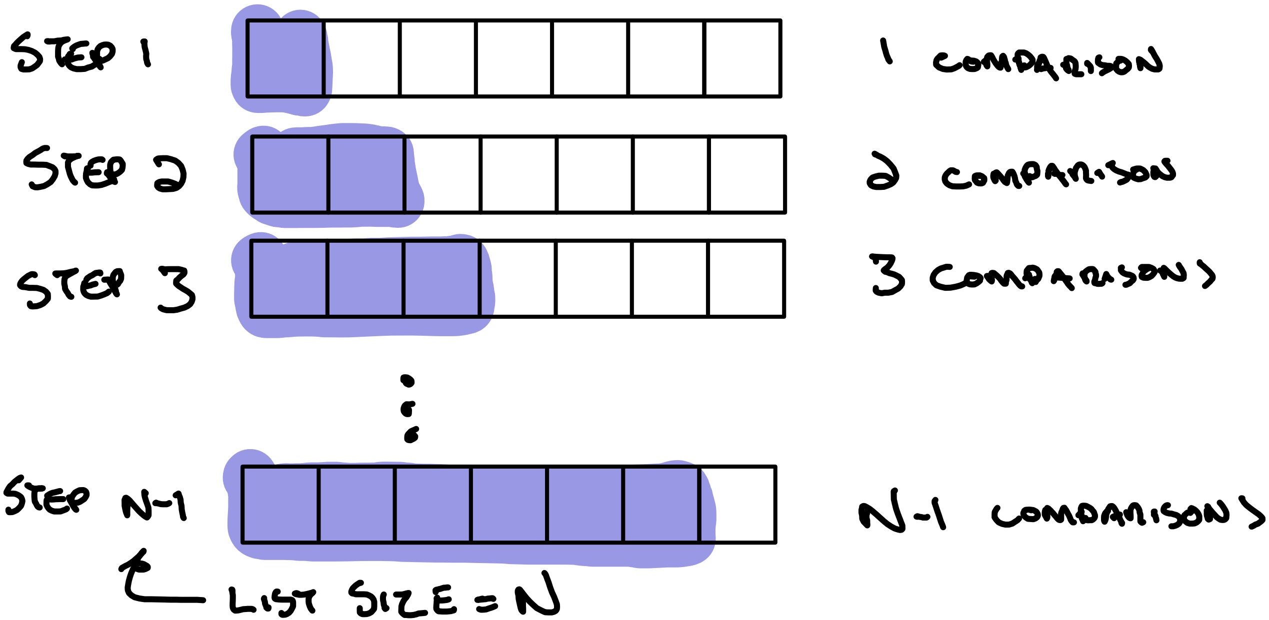
(I'd love to take a response from you all to do with the cards, if you'd like please build your example with values 2,3,4,5,6)

best case: reverse sorted list (6,5,4,3,2). each new item is added to sorted sub list with 1 comparison

worst case: sorted list (2,3,4,5,6). each new item has to be compared to all items currently in sub list

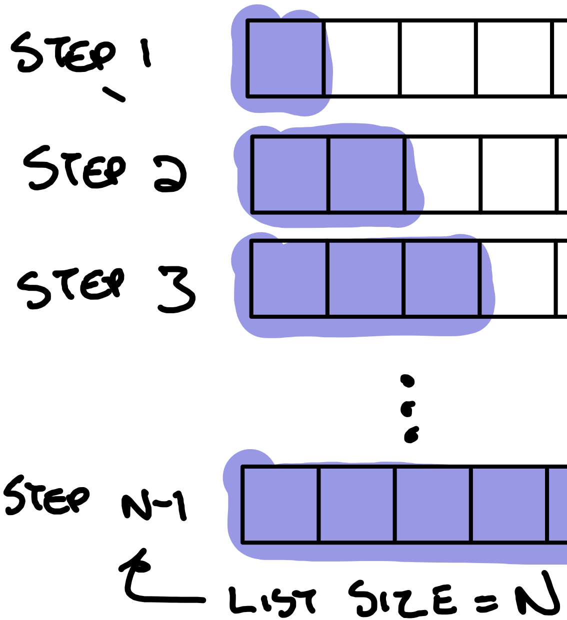
Worst Case Analysis: Insertion Sort

In the worst case, each new item must be compared to all the previously sorted items.



Worst Case Analysis: Insertion Sort

In the worst case, each new item must be compared to all the previously sorted items.



$$1+2+3+\dots+N-1$$

$$=\sum_{k=1}^{N-1} k$$

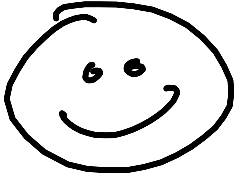
$$=\frac{1+(N-1)}{2}\cdot N$$

$$=\frac{N^2}{2}-\frac{N}{2}=O(N^2)$$

NOTICE IN LAST SLIDE:

ALL THAT MATH WE DID TOGETHER

REALLY DOES WORK TOGETHER!



(IT DOES TAKE ALMOST WHOLE SEMESTER THOUGH...)

ON OUR NEXT|LAST EPISODE OF CS1800 ...

CAN WE SORT FASTER THAN $O(n^2)$?