Linear Search

```
def linear_search(v,b):
    """Returns: first occurrence of v in b (-1 if not found)
    Precond: b a list of number, v a number
    """
    # Loop variable
    i = 0
    while i < len(b) and b[i] != v:
        i = i + 1

    if i == len(b):  # not found
        return -1
    return i
```

How many entries do we have to look at?

All of them!

1

Binary Search

```
def binary_search(v,b):
    # Loop variable(s)
    i = 0, j = len(b)
    while i < j and b[i] != v:
        mid = (i+j)//2
        if b[mid] < v:
            j = mid
        elif b[mid] > v:
            i = mid
        else:
            return mid
    return -1 # not found
```

Requires that the data is sorted!

But few checks!

2

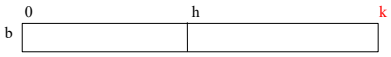
The Sorting Challenge

- **Given:** A list of numbers
- **Goal:** Sort those numbers using only
  - Iteration (while-loops or for-loops)
  - Comparisons (< or >)
  - Assignment statements
- Why? For proper **analysis**.
  - Methods/functions come with hidden costs
  - Everything above has no hidden costs
  - Each comparison or assignment is "1 step"

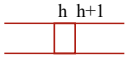
3

Horizontal Notation

- Want a pictoral way to visualize this sorting
  - Represent the list as long rectangle
  - We saw this idea in divide-and-conquer



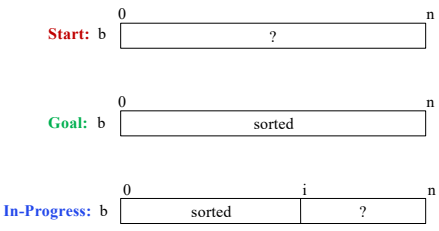
- Do **not** show individual boxes
  - Just dividing lines between regions
  - Label dividing lines with indices
  - But index is either left or right of dividing line



(h+1) – h = 1

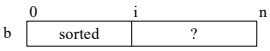
4

Visualizing Sorting

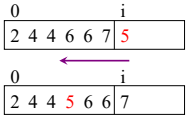


5

Insertion Sort




```
i = 0
while i < n:
    # Push b[i] down into its
    # sorted position in b[0..i]
    i = i+1
```



Remember the restrictions!

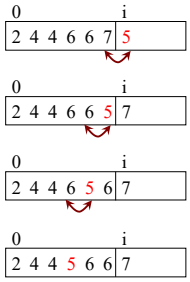
6

Insertion Sort: Moving into Position

```
i = 0
while i < n:
    push_down(b,i)
    i = i+1

def push_down(b, i):
    j = i
    while j > 0:
        if b[j-1] > b[j]:
            swap(b,j-1,j)
        j = j-1
```

swap shown in the lecture about lists



7

Insertion Sort: Performance

```
def push_down(b, i):
    """Push value at position i into
    sorted position in b[0..i-1]"""
    j = i
    while j > 0:
        if b[j-1] > b[j]:
            swap(b,j-1,j)
        j = j-1
```

Insertion sort is an n^2 algorithm

- b[0..i-1]: i elements
- Worst case:
  - i = 0: 0 swaps
  - i = 1: 1 swap
  - i = 2: 2 swaps
- Pushdown is in a loop
  - Called for i in 0..n
  - i swaps each time

Total Swaps: $0 + 1 + 2 + 3 + \ldots (n-1) = (n-1)*n/2 = (n^2-n)/2$

8

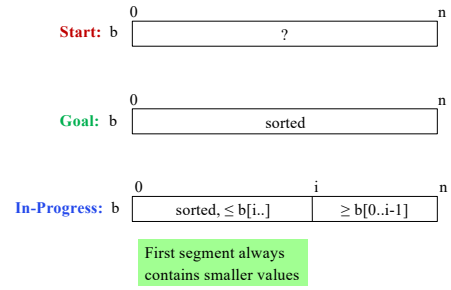
Algorithm "Complexity"

- **Given**: a list of length n and a problem to solve
- **Complexity**: *rough* number of steps to solve worst case
- Suppose we can compute 1000 operations a second:

Complexity	n=10	n=100	n=1000
log n	0.003 s	0.006 s	0.01 s
n	0.01 s	0.1 s	1 s
n log n	0.016 s	0.32 s	4.79 s
n^2	0.1 s	10 s	16.7 m
n^3	1 s	16.7 m	11.6 d
2^n	1 s	$4x10^{19}$ y	$3x10^{290}$ y

9

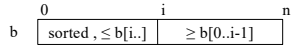
A New Algorithm



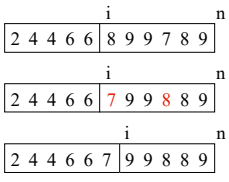
First segment always contains smaller values

10

Selection Sort




```
i = 0
while i < n:
    # Find minimum in b[i..]
    # Move it to position i
    i = i+1
```



Remember the restrictions!

11

What is the Problem?

- Both insertion, selection sort are **nested loops**
  - **Outer loop** over each element to sort
  - **Inner loop** to put next element in place
  - Each loop is n steps. $n \times n = n^2$
- To do better we must *eliminate* a loop
  - But how do we do that?
  - What is like a loop? **Recursion!**
  - Will see how to do this next lecture

12