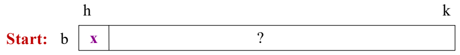
Recall Our Problem

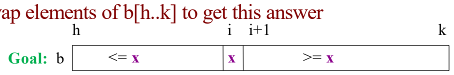
- Both insertion, selection sort are **nested loops**
  - **Outer loop** over each element to sort
  - **Inner loop** to put next element in place
  - Each loop is n steps. n × n = n^2
- To do better we must *eliminate* a loop
  - But how do we do that?
  - What is like a loop? **Recursion!**
  - First need an *intermediate* algorithm

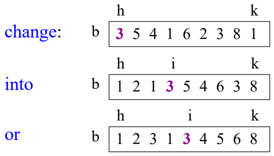
The Partition Algorithm

- Given a list segment b[h..k] with some value x in b[h]:



- Swap elements of b[h..k] to get this answer

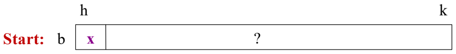




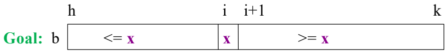
- x is called the pivot value
  - x is not a program variable
  - denotes value initially in b[h]

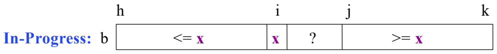
Designing the Partition Algorithm

- Given a list b[h..k] with some value x in b[h]:



- Swap elements of b[h..k] to get this answer





Indices b, h important!
Might partition only part

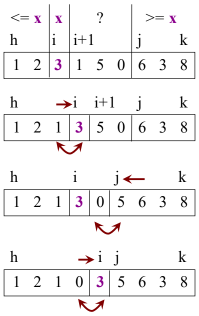
Implementating the Partition Algorithm

```
def partition(b, h, k):
    """Partition list b[h..k] around a pivot x = b[h]"""
    i = h; j = k+1; x = b[h]

    while i < j-1:
        if b[i+1] >= x:
            # Move to end of block.
            swap(b,i+1,j-1)
            j = j - 1
        else:   # b[i+1] < x
            swap(b,i,i+1)
            i = i + 1

    return i
```

partition(b,h,k), not partition(b[h:k+1])
Remember, slicing always copies the list!
We want to partition the **original** list

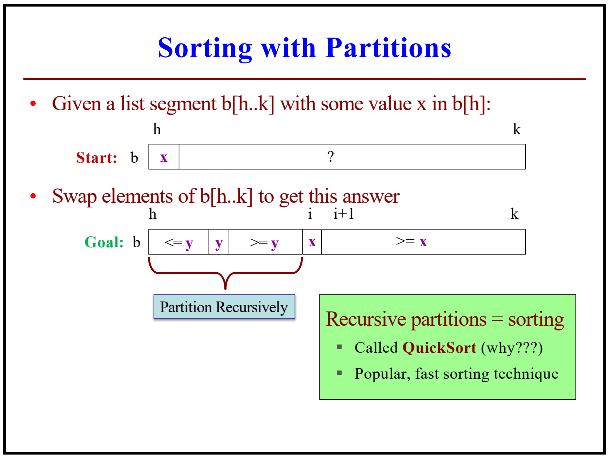
Partition Algorithm Implementation

```
def partition(b, h, k):
    """Partition list b[h..k] around a pivot x = b[h]"""
    i = h; j = k+1; x = b[h]

    while i < j-1:
        if b[i+1] >= x:
            # Move to end of block.
            swap(b,i+1,j-1)
            j = j - 1
        else:   # b[i+1] < x
            swap(b,i,i+1)
            i = i + 1

    return i
```



Why is this Useful?

- Will use this algorithm to replace inner loop
  - The inner loop cost us n swaps every time
- Can this reduce the number of swaps?
  - Worst case is k-h swaps
  - This is n if partitioning the whole list
  - But less if only partitioning part
- **Idea:** Break up list and partition only part?
  - This is **Divide-and-Conquer!**



Sorting with Partitions

- Given a list segment b[h..k] with some value x in b[h]:

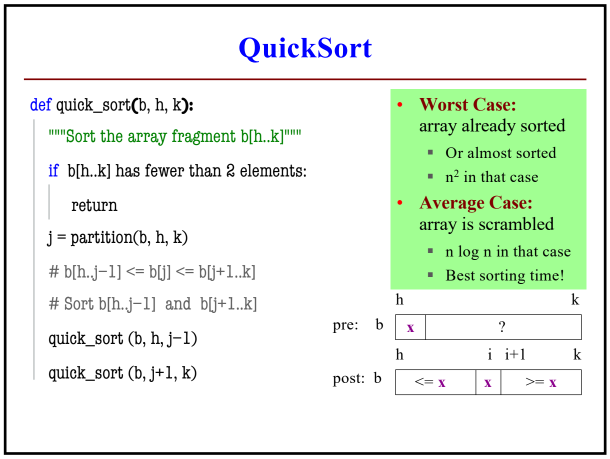
h k
Start: b x ?

- Swap elements of b[h..k] to get this answer

h i i+1 k
Goal: b <= y y >= y x >= x

Partition Recursively

Recursive partitions = sorting
  - Called **QuickSort** (why???)
  - Popular, fast sorting technique



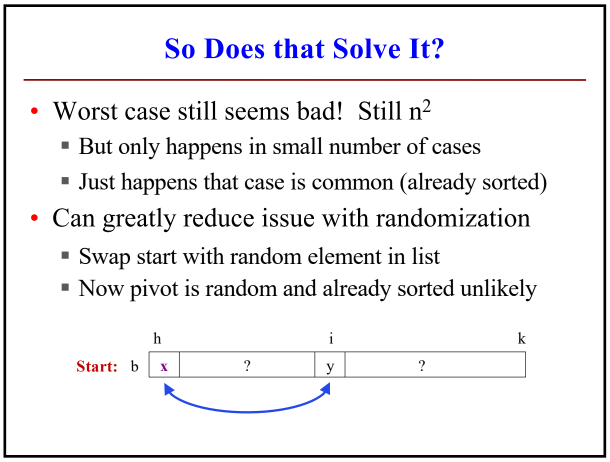
QuickSort

```
def quick_sort(b, h, k):
    """Sort the array fragment b[h..k]"""
    if b[h..k] has fewer than 2 elements:
        return
    j = partition(b, h, k)
    # b[h..j-1] <= b[j] <= b[j+1..k]
    # Sort b[h..j-1]  and  b[j+1..k]
    quick_sort (b, h, j-1)
    quick_sort (b, j+1, k)
```

- **Worst Case:** array already sorted
  - Or almost sorted
  - n^2 in that case
- **Average Case:** array is scrambled
  - n log n in that case
  - Best sorting time!

h k
pre: b x ?

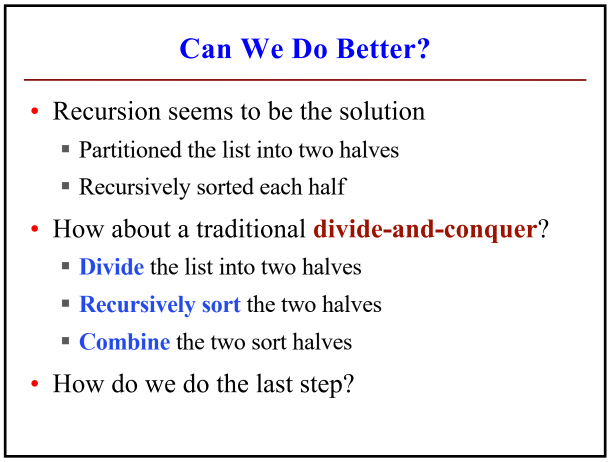
h i i+1 k
post: b <= x x >= x



So Does that Solve It?

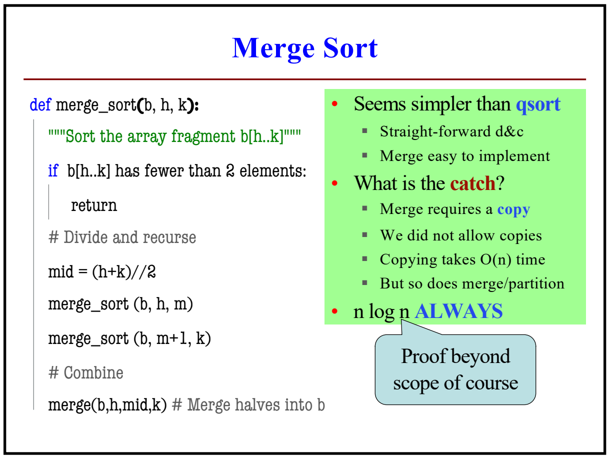
- Worst case still seems bad! Still n^2
  - But only happens in small number of cases
  - Just happens that case is common (already sorted)
- Can greatly reduce issue with randomization
  - Swap start with random element in list
  - Now pivot is random and already sorted unlikely

h i k
Start: b x ? y ?



Can We Do Better?

- Recursion seems to be the solution
  - Partitioned the list into two halves
  - Recursively sorted each half
- How about a traditional **divide-and-conquer**?
  - **Divide** the list into two halves
  - **Recursively sort** the two halves
  - **Combine** the two sort halves
- How do we do the last step?

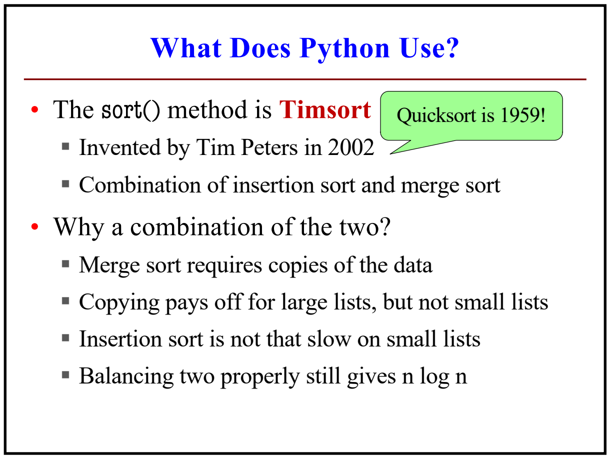


Merge Sort

```
def merge_sort(b, h, k):
    """Sort the array fragment b[h..k]"""
    if b[h..k] has fewer than 2 elements:
        return
    # Divide and recurse
    mid = (h+k)//2
    merge_sort (b, h, m)
    merge_sort (b, m+1, k)
    # Combine
    merge(b,h,mid,k) # Merge halves into b
```

- Seems simpler than **qsort**
  - Straight-forward d&c
  - Merge easy to implement
- What is the **catch**?
  - Merge requires a **copy**
  - We did not allow copies
  - Copying takes O(n) time
  - But so does merge/partition
- n log n **ALWAYS**

Proof beyond scope of course



What Does Python Use?

- The sort() method is **Timsort**
  - Invented by Tim Peters in 2002
  - Combination of insertion sort and merge sort
- Why a combination of the two?
  - Merge sort requires copies of the data
  - Copying pays off for large lists, but not small lists
  - Insertion sort is not that slow on small lists
  - Balancing two properly still gives n log n

Quicksort is 1959!