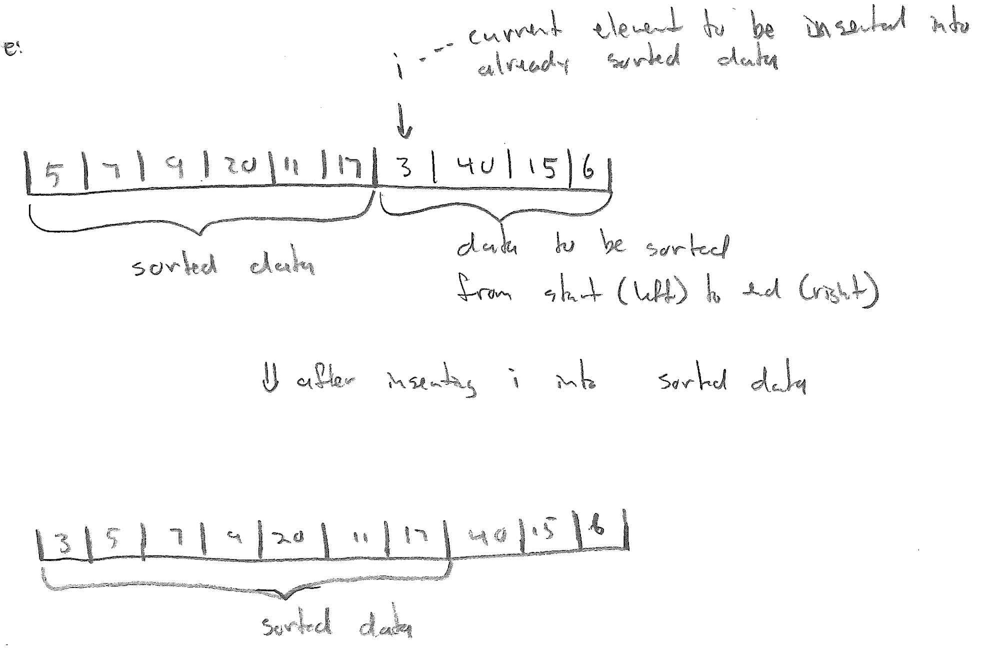
Sorting

- Often need efficient algorithms to order data within your program
- Comparison sorts rely on comparing elements to determine the order for those elements
- Data can be sorted in ascending order or descending order

Insertion Sort: Data is sorted by examining each element from the beginning to end, where each element is inserted into the correct location within the already sorted data

Example:


i --- current element to be inserted into already sorted data
| 5 | 7 | 9 | 20 | 11 | 17 | 3 | 40 | 15 | 6 |
sorted data
data to be sorted from start (left) to end (right)
⇓ after inserting i into sorted data
| 3 | 5 | 7 | 9 | 20 | 11 | 17 | 40 | 15 | 6 |
sorted data

Insertion Sort Pseudocode:

```
① for j=1 to size-1                          } loops over n-1 elements
    a) i = j-1
    b) key = a[j]
    c) while i >= 0 and a[i] > key           } in worst case you must examine
        I) a[i+1] = a[i]                       all previous elements
        II) i--
    d) a[i+1]=key
```

Run time $T(n) = \sum_{i=1}^{n-1} i$

$= 1 + 2 + 3 + 4 + 5 + 6 + 7 + \ldots + n-1$

$$T(n) = \frac{n(n+1)}{2} - n$$

$$O\left(\frac{n(n+1)}{2} - n\right) = \underline{\underline{O(n^2)}}$$

Benefits of Insertion Sort: Inserting a new item into a sorted array using insertion sort has a complexity of O(n)

↳ * Let's see bubblesort do that!

Quicksort: Recursively partions data and sorts each partition

- Partitioning is done by selecting an element (often called the pivot) unsorted data. All elements greater than pivot will be moved to the right of pivot. All elements less than pivot will be moved to the left of pivot.

- How to choose the pivot?

  ↳ Many options exist (purely random, median, median-of three)

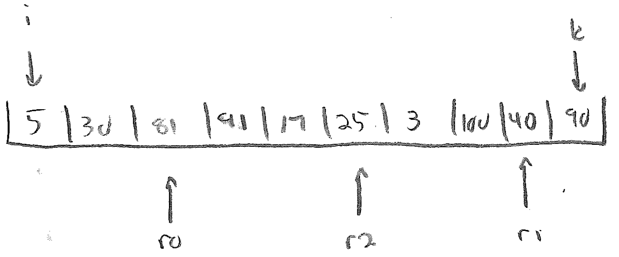
  ↳ Median of three:

  ① pick three random locations within array

  ② choose the median of the three

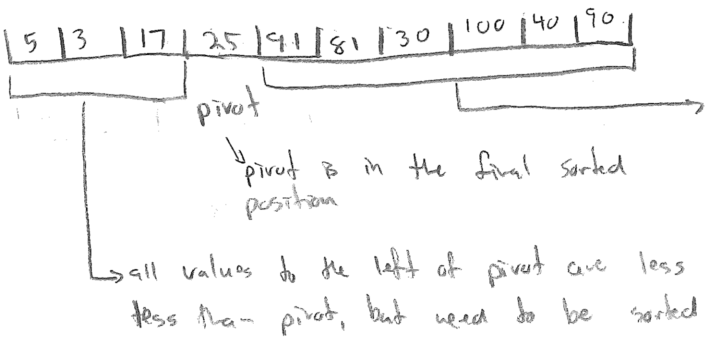
- After partitioning we need to know the location of the pivot.

Example:



median = 25

⇓ after partitioning



pivot

↳ pivot is in the final sorted position

all values right of pivot are greater than pivot, but need to be sorted

all values to the left of pivot are less less than pivot, but need to be sorted

Pseudocode for Partition operator:

```
int partition (int * data, int i, int k)
    ① r0 = (rand() % (k-i+1)) + i          ] - find three random locations between
    ② r1 = (rand() % (k-i+1)) + i          ]   i and k inclusive
    ③ r2 = (rand() % (k-i+1)) + i          ]
    ④ pval = median (data[r0], data[r1], data[r2])          *this is the pivot value
    ⑤ i--
    ⑥ k++
    ⑦ while (1) {
        a) do
            [i] k--
        b) while (a[k] > pval)
        c) do
            [i] i++
        d) while (a[i] < pval)
        e) if i < k
            [i] swap (a[i], a[k])
        f) else return k
```

Quicksort Pseudocode

```
qksort (int *data, int i, int k)

1 if (i < k)
    a) j = partition (data, i, k)
       qksort (data, i, j)        * this change is important!
       qksort (data, j+1, k)
```

Initial call to Quicksort: qksort (data, 0, size-1)

Runtime Complexity:

1. Worst-case: Occurs when result of each partitioning has one subproblem of size 0 and one subproblem of size n-1

   ↳ Partitioning will repeat n times each with a subproblem 1 smaller

   ⇓

   * Note: This is similar to how insertion sort works

$$T(n) = O(n) + T(n-1)$$

$$= n + n-1 + n-2 + n-3 + \cdots + 1$$

$$\Theta(n^2)$$

② Best Case: Occurs when as even a split possible is achieved, resulting in two subproblems of size $\frac{n}{2}$

$$T(n) = 2T\left(\frac{n}{2}\right) + O(n)$$

$$= n + 2\left(\frac{n}{2}\right) + 2\left(\frac{n}{4}\right) + 2\left(\frac{n}{8}\right) + \cdots + 1$$

$O(n \log n)$

③ Average Case: $O(n \log n)$

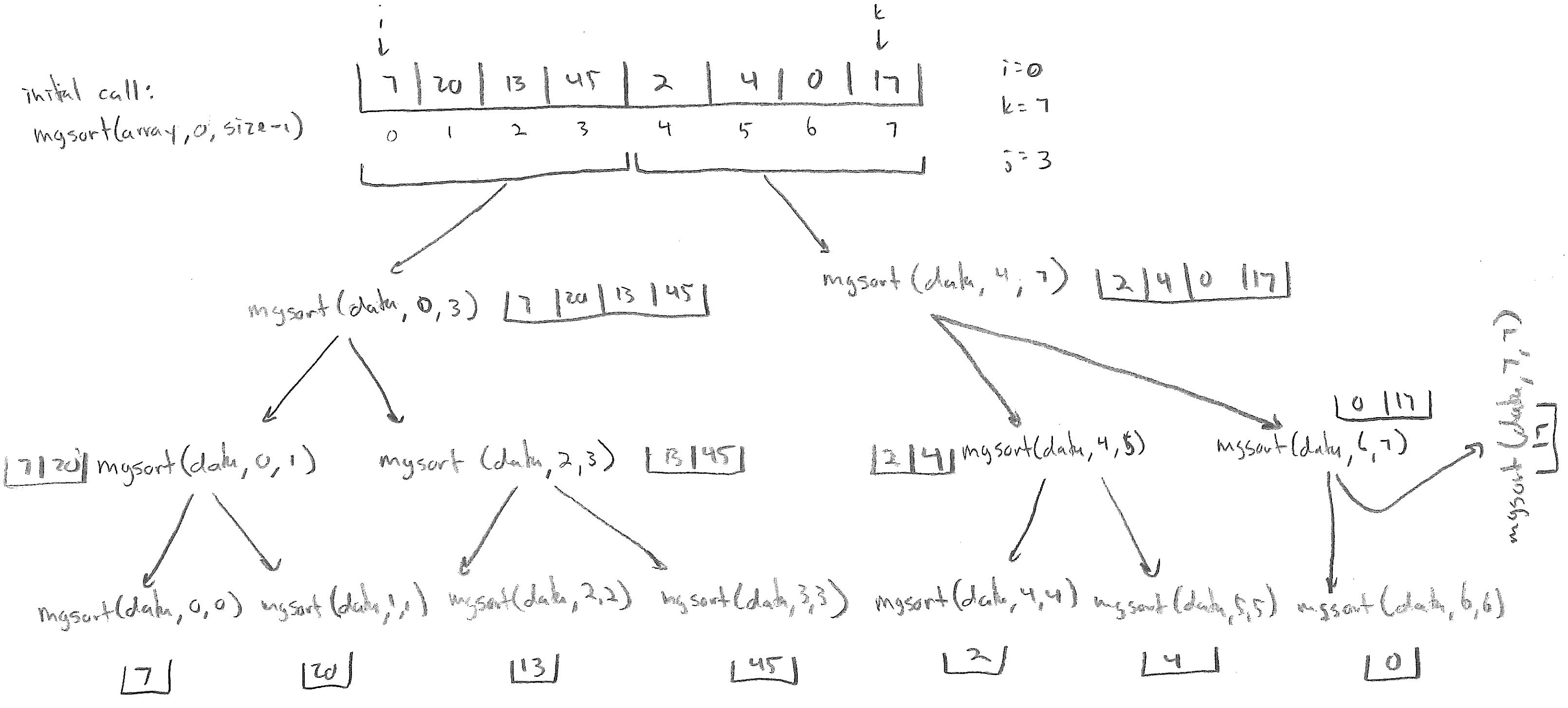
Mergesort: Sorting method that divedes data in half at each stage until a single node is found.(which is therefore sorted), and merges the data within two halves into a sorted array

- + Merging process can be performed efficiently as a single pass over the data
- + Efficient algorithm with worst case complexity of O(n log n)
- + Potential drawback is the need for additional memory as the merge cannot be performed in place

Pseudocode for Mergesort:

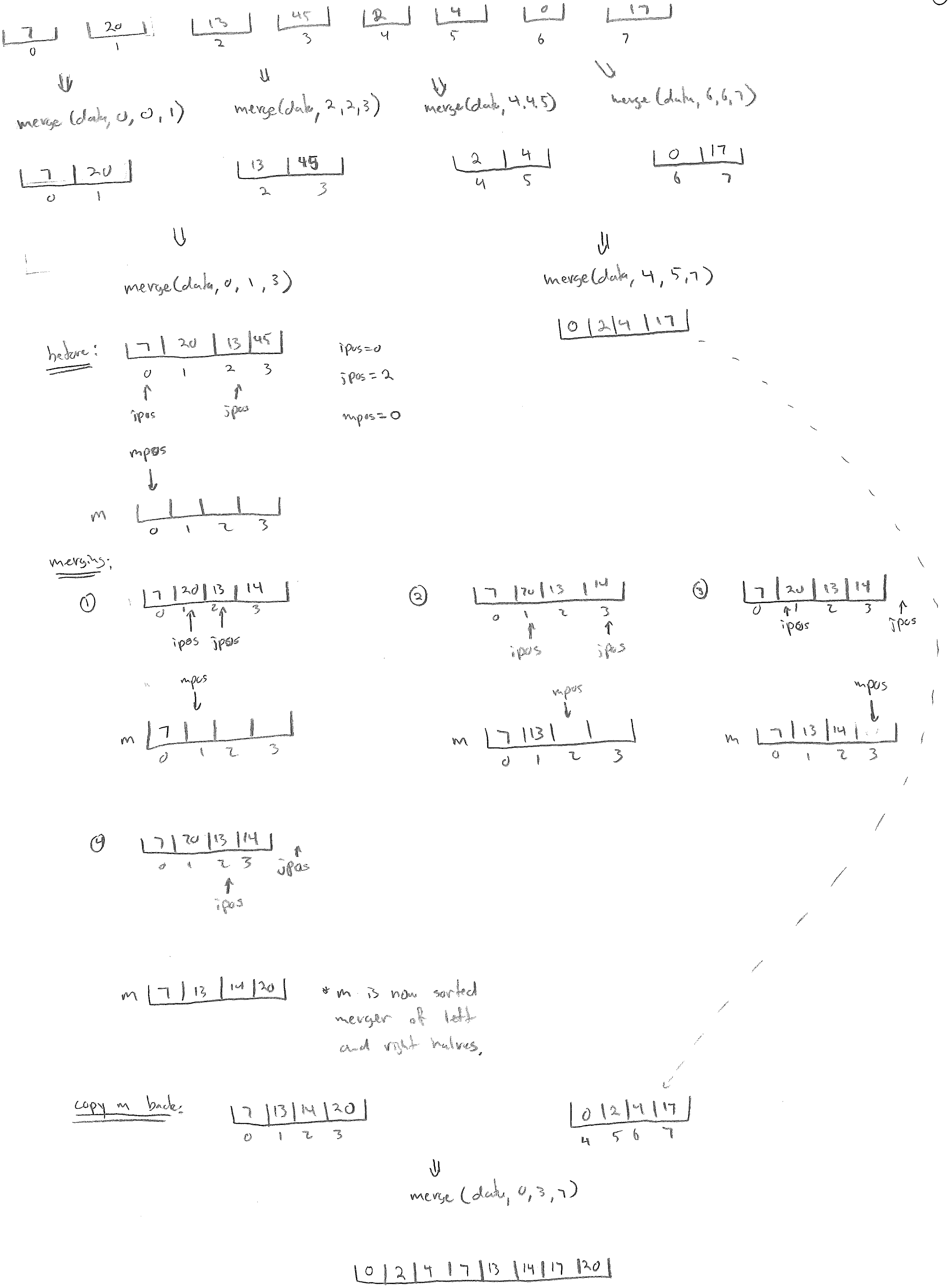
```
int mgsort(int *data, int i, int k)
    ① if i < k                    * we have at least two items
                                    to sort
        a) j = (i+k-1)/2
        b) mgsort(data, i, j)
        c) mgsort(data, j+1, k)
        d) merge(data, i, j, k)
```

*i and k are the low and high indices within the array currently being sorted



Pseudocode for merge process

```
merge (ind * data, int i, int j, int k)

  ① ipos = i                 * next location in left half
  ② jpos = j+1               * next location in right half
  ③ mpos = 0                 * next location in merged array

  ④ allocate storage for merge elements
       m = (int *) malloc( sizeof(int) * (k - i + 1))

  ⑤ while ( ipos <= j || jpos <= k)

      a) if ipos > j                   * no elements left in left half

         I) while jpos <= k

              i   m[mpos] = data[jpos]
              ii  jpos++
              iii mpos++
         II) break
      b) else if jpos > k              * no elements left in left half

         I) while ipos <= j

              i   m[pos] = data[ipos]
              ii  ipos++
              iii mpos++

         II) break

      c) if data[ipos] < data[jpos]

         I) m[mpos] = data[ipos]
         II) ipos++
         III) mpos++

      d) else
         I) m[mpos] = data[jpos]
         II) jpos++
         III) mpos++

  ⑥ copy m back to data
       memcpy (& data[i], m, sizeof(int) * (k - i + 1))

  ⑦ free m

  ⑧ return success
```



* m is now sorted merger of left and right halves.