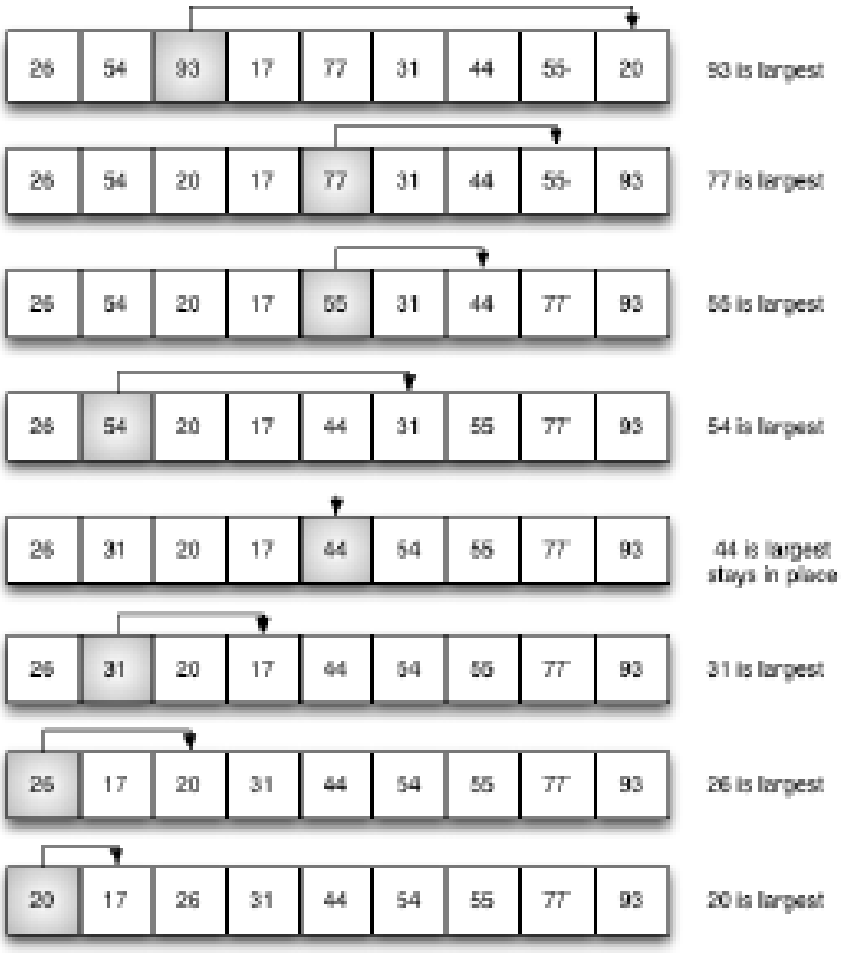
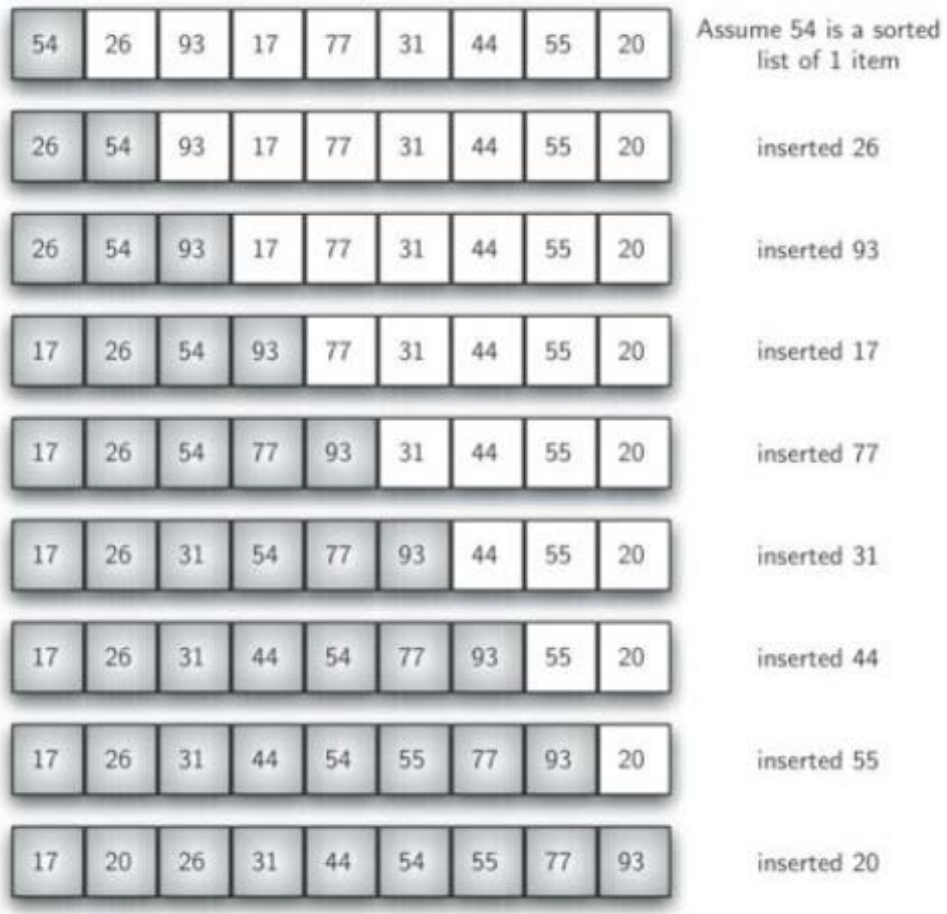
Selection Sort



Selection Sort

```
#ArrayLst -> ArrayLst
#sort using selection sort
def selection_sort(lst):
        lst_size = lst.size
        i = 0
        while i < lst_size-1:
                maxIndex = 0
                j = 1
                while j < lst_size-1-i:
                        if lst.value[j] > lst.value[maxIndex]:
                                maxIndex = j
                        j += 1
                temp = lst.value[lst_size-1-i]
                lst.value[lst_size-1-i] = lst.value[maxIndex]
                lst.value[maxIndex] = temp
                i += 1
```

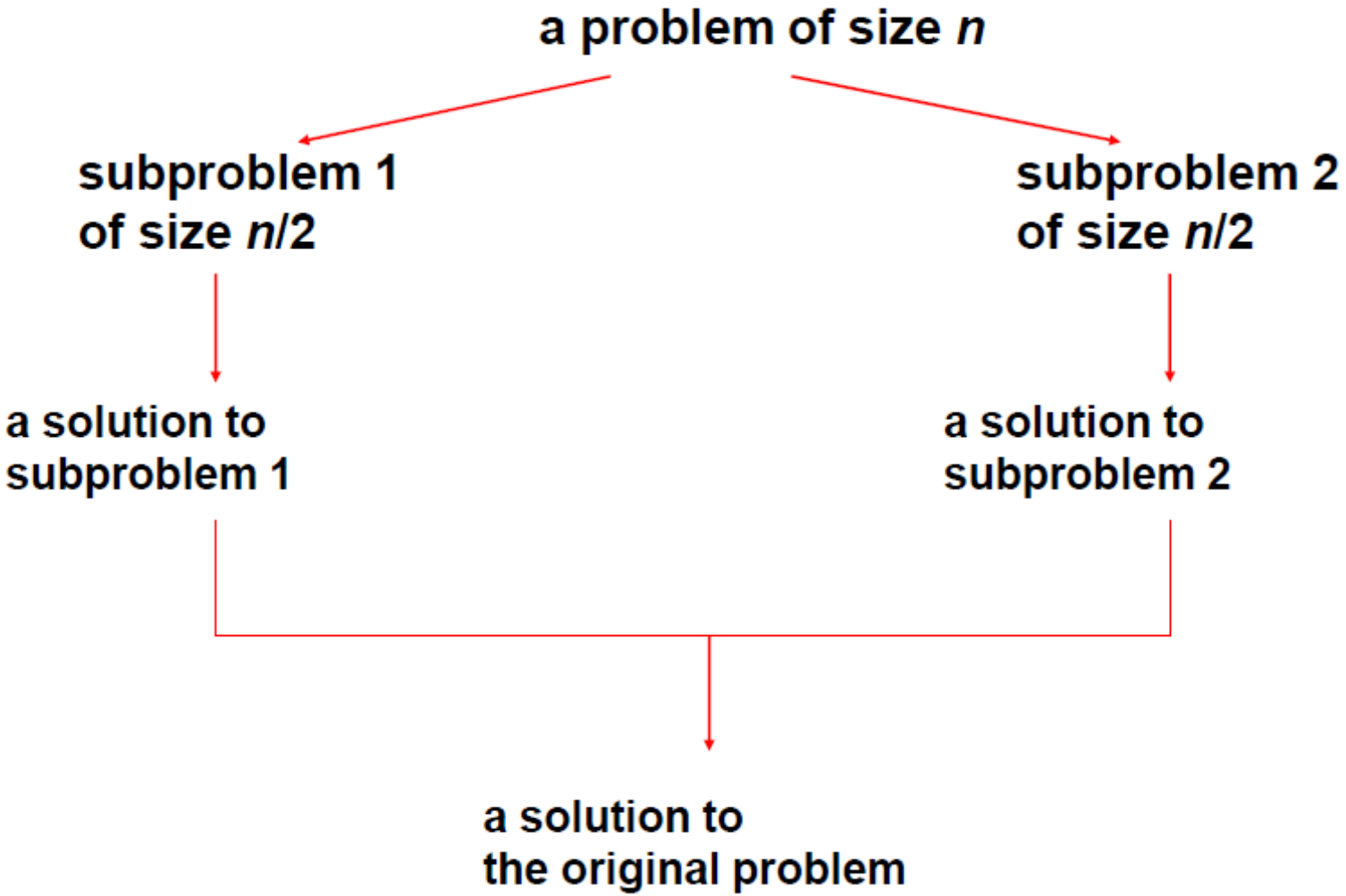
Insertion Sort



Insertion Sort

```
#ArrayLst -> ArrayLst
#sort using insertion sort
def insertion_sort(lst):
    lst_size = lst.size
    swapped = 1
    i = 0
    while i < lst_size-1 and swapped == 1:
        swapped = 0
        j=0
        while j < lst_size-i-1:
            if lst.value[j] > lst.value[j+1]:
                temp = lst.value[j]
                lst.value[j] = lst.value[j+1]
                lst.value[j+1] = temp
                swapped = 1
            j += 1
        i += 1
    return lst
```

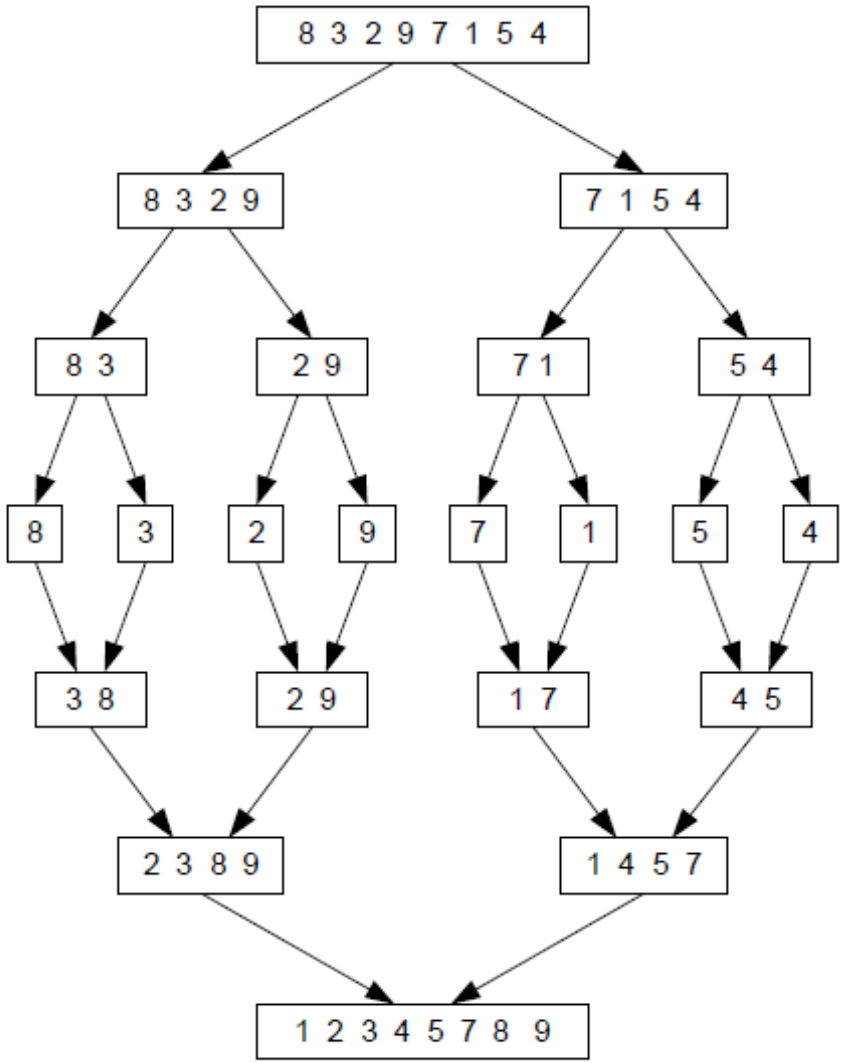
Divide-and-Conquer Technique



Mergesort

- Split array A[0..n-1] in two about equal halves and make copies of each half in arrays B and C
- Sort arrays B and C recursively
- Merge sorted arrays B and C into array A as follows:
  - Repeat the following until no elements remain in one of the arrays:
    - » compare the first elements in the remaining unprocessed portions of the arrays
    - » copy the smaller of the two into A, while incrementing the index indicating the unprocessed portion of that array
  - Once all elements in one of the arrays are processed, copy the remaining unprocessed elements from the other array into A.

Mergesort Example



Mergesort

```
def mergesort(list):
    if (len(list) < 2):
        return list

    middle = len(list)//2
    left = mergesort(list[:middle])
    right = mergesort(list[middle:])

    return merge(left, right)
```

```
def merge(left, right):
    print("\nleft:", left, "\nright:", right)
    result = []
    i, j = 0, 0
    while (len(result) < len (left) + len(right)):
        if left[i] < right[j]:
            result.append(left[i])
            i += 1
        else:
            result.append(right[j])
            j += 1
        if i == len(left):
            result.extend(right[j:])
        if j == len(right):
            result.extend(left[i:])
    print("result", result)
    return result
```

Quicksort

- Fastest known sorting algorithm in practice
  - Caveats: not stable
- Average case complexity → $O(N \log N)$
- Worst-case complexity → $O(N^2)$
  - Rarely happens, if coded correctly

Quicksort Outline

- Divide and conquer approach
- Given array S to be sorted
  - If size of S < 1 then done;
  - Pick any element v in S as the pivot
  - Partition S-{v} (remaining elements in S) into two groups
    - » S1 = {all elements in S-{v} that are ≤ than v}
    - » S2 = {all elements in S-{v} that are ≥ than v}
  - Return {quicksort(S1) followed by v followed by quicksort(S2) }
- Trick lies in handling the partitioning (step 3).
  - Picking a good pivot
  - Efficiently partitioning in-place

Picking the Pivot

Strategy 1: Pick the first element in S

- Works only if input is random
- What if input S is sorted, or even mostly sorted?
  - All the remaining elements would go into either S1 or S2!
  - Terrible performance!
- Why worry about sorted input?
  - Remember → Quicksort is recursive, so sub-problems could be sorted
  - Plus mostly sorted input is quite frequent

Picking the Pivot (contd.)

Strategy 2: Pick the pivot randomly

- Would usually work well, even for mostly sorted input
- Unless the random number generator is not quite random!
- Plus random number generation is an expensive operation

Picking the Pivot (contd.)

Strategy 3: Median-of-three Partitioning

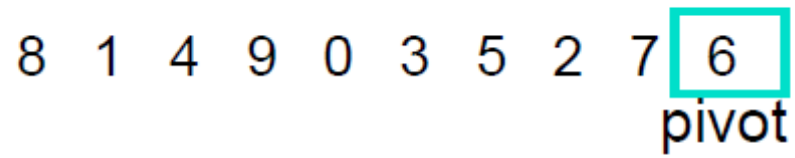
- *Ideally*, the pivot should be the median of input array **s**
  - Median = element in the middle of the sorted sequence
- Would divide the input into two almost equal partitions
- But, its hard to calculate median quickly, without sorting first!
- So find the approximate median
  - Pivot = median of the left-most, right-most and center element of the array **s**
  - Solves the problem of sorted input

Picking the Pivot (contd.)

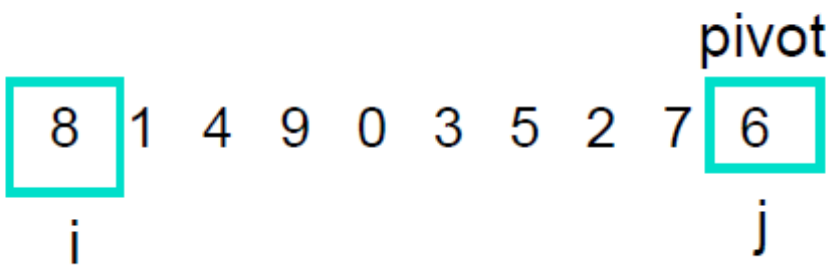
- Example: Median-of-three Partitioning
  - Let input **`S = {6, 1, 4, 9, 0, 3, 5, 2, 7, 8}`**
  - **`left=0`** and **`S[left] = 6`**
  - **`right=9`** and **`S[right] = 8`**
  - **`center = (left+right)/2 = 4`** and **`S[center] = 0`**
  - Pivot
    - » = Median of **`S[left]`**, **`S[right]`**, and **`S[center]`**
    - » = median of 6, 8, and 0
    - » **`= S[left] = 6`**

Partitioning Algorithm Idea

- Original input : `S = {6, 1, 4, 9, 0, 3, 5, 2, 7, 8}`
- Get the pivot out of the way by swapping it with the last element



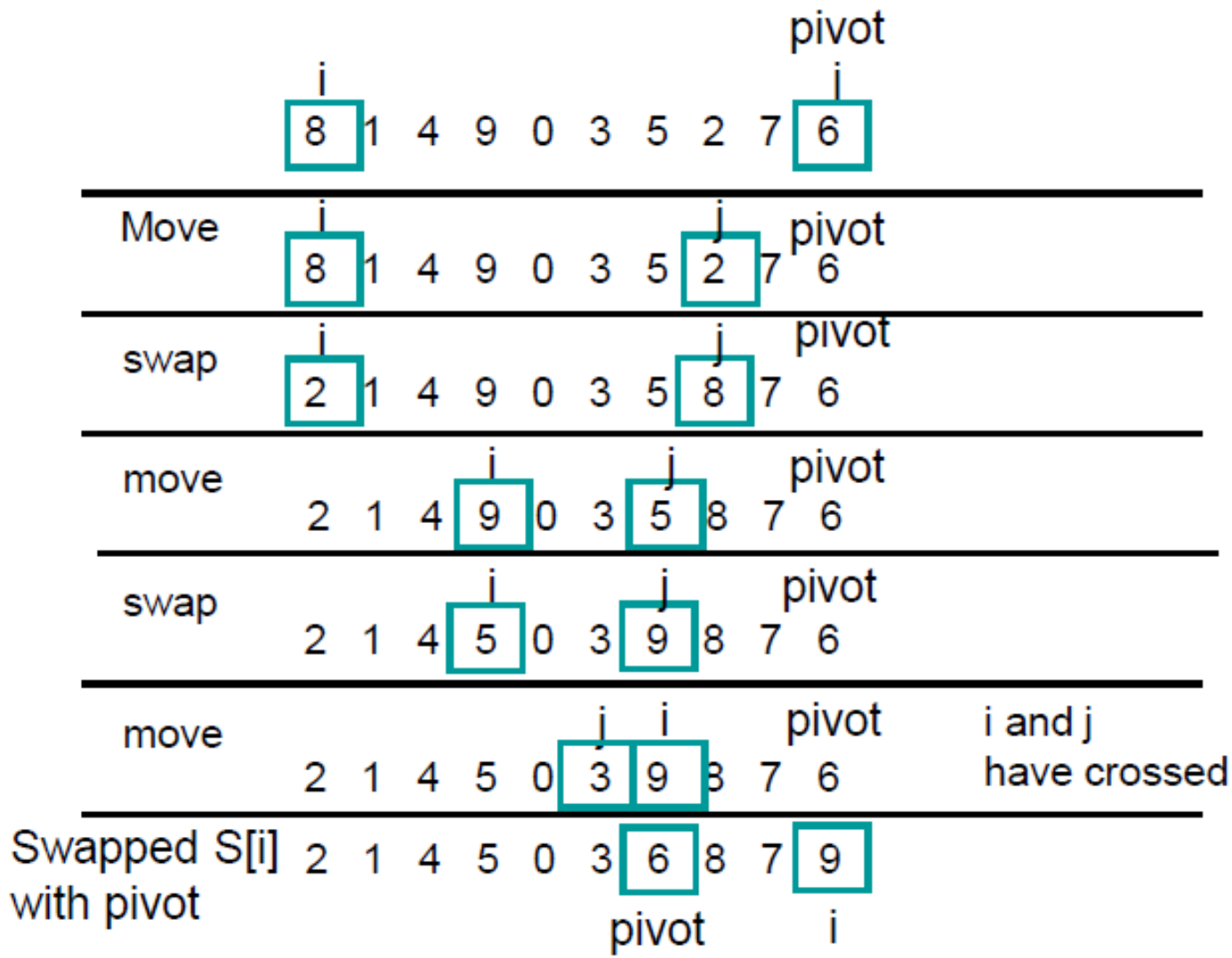
- Have two 'iterators' – `i` and `j`
  - `i` starts at first element and moves forward
  - `j` starts at last element and moves backwards



Partitioning Algorithm Idea (contd.)

- **`While (i < j)`**
  - Move **`i`** to the right till we find a number ≥ than **`pivot`**
  - Move **`j`** to the left till we find a number ≤ than **`pivot`**
  - **`If (i < j) swap(S[i], S[j])`**
- Swap the **`pivot`** with **`S[i]`**

Partitioning Algorithm Illustrated



Quicksort

```
def quickSort(alist):
    quickSortHelper(alist,0,len(alist)-1)

def quickSortHelper(alist,first,last):
    if first<last:

        splitpoint = partition(alist,first,last)

        quickSortHelper(alist,first,splitpoint-1)
        quickSortHelper(alist,splitpoint+1,last)
```

```
def partition(alist,first,last):
    pivotvalue = alist[first]
    print("list start:", alist)
    print("pivot:", pivotvalue)

    leftmark = first+1
    rightmark = last

    done = False
    while not done:

        while leftmark <= rightmark and alist[leftmark] <= pivotvalue:
            leftmark = leftmark + 1

        while alist[rightmark] >= pivotvalue and rightmark >= leftmark:
            rightmark = rightmark -1

        if rightmark < leftmark:
            done = True
        else:
            temp = alist[leftmark]
            alist[leftmark] = alist[rightmark]
            alist[rightmark] = temp
```

Just choosing first element as pivot
Could use median of three, random or other method

Dealing with small arrays

- For small arrays `(N ≤ 20)`,
  - Insertion sort is faster than quicksort
- Quicksort is recursive
  - So it can spend a lot of time sorting small arrays
- Hybrid algorithm:
  - Switch to using insertion sort when problem size is small (say for `N < 20`)