Introduction to Computer Science:

Algorithm - Sorting

April 2020

Honguk Woo

Quiz : Sum of pair

- Given an array a[] and a number x, check for pair in a[] with sum as x

  a = {1, 4, 45, 6, 10, -8} and sum = 16

- Complexity of Exhaustive Search ?

- Better solution ?

Sorting

- Sorting : Arranging items in a collection so that there is an ordering on one (or more) of the fields in the items
- So many solutions for it
  - O(n^2), O($n$lg n), O(n)
  - depending on
    - simplicity of mind
    - complexity of insert operation
  - Big O notation (e.g., O(n^2)) is used in Computer Science to describe the performance or complexity of an algorithm.

Sorting

- **Sorting algorithms :** Algorithms that order the items in the collection based on the sort key
  - Sort Key : the field (or fields) on which the ordering is based

- Comparison-based algorithms
  - Selection
  - Insertion
  - Bubble
  - Merge
  - Quick

  :

  :

Selection Sort

- Maintain two parts : sorted, unsorted
- Selection : in every iteration i, select the minimum (maximum) from the unsorted part, and move it to the sorted part (current index i)
- Two for-loops:
  - i = 0….n-2
  - For ith iteration, j = i+1…n-1
  - Keep the index of minimum
- Comparisons
  - (n - 1) + (n - 2) + … + 1 = n(n - 1)/2

arr[] = 64 25 12 22 11

// Find the minimum element in arr[0...4]
// and place it at beginning
11 25 12 22 64

// Find the minimum element in arr[1...4]
// and place it at beginning of arr[1...4]
11 **12** 25 22 64

// Find the minimum element in arr[2...4]
// and place it at beginning of arr[2...4]
11 12 **22** 25 64

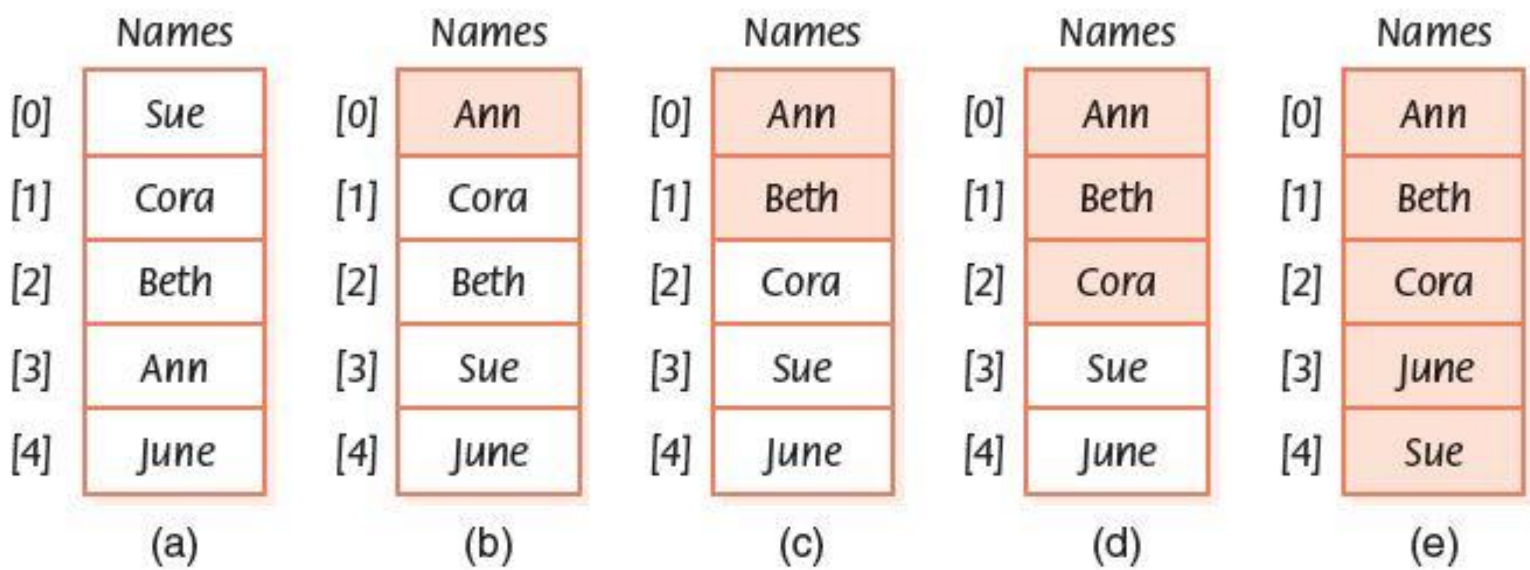
// Find the minimum element in arr[3...4]
// and place it at beginning of arr[3...4]
11 12 22 **25** 64

Selection Sort

```
Selection Sort

Set firstUnsorted to 0
WHILE (  1. not sorted yet  )
     2. Find smallest unsorted item
     3. Swap firstUnsorted item with the smallest
     Set firstUnsorted to firstUnsorted + 1
```

Selection Sort

- Alphabetical order case

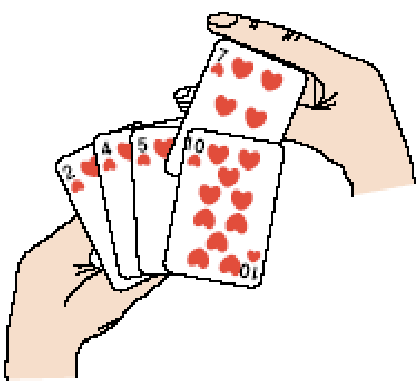


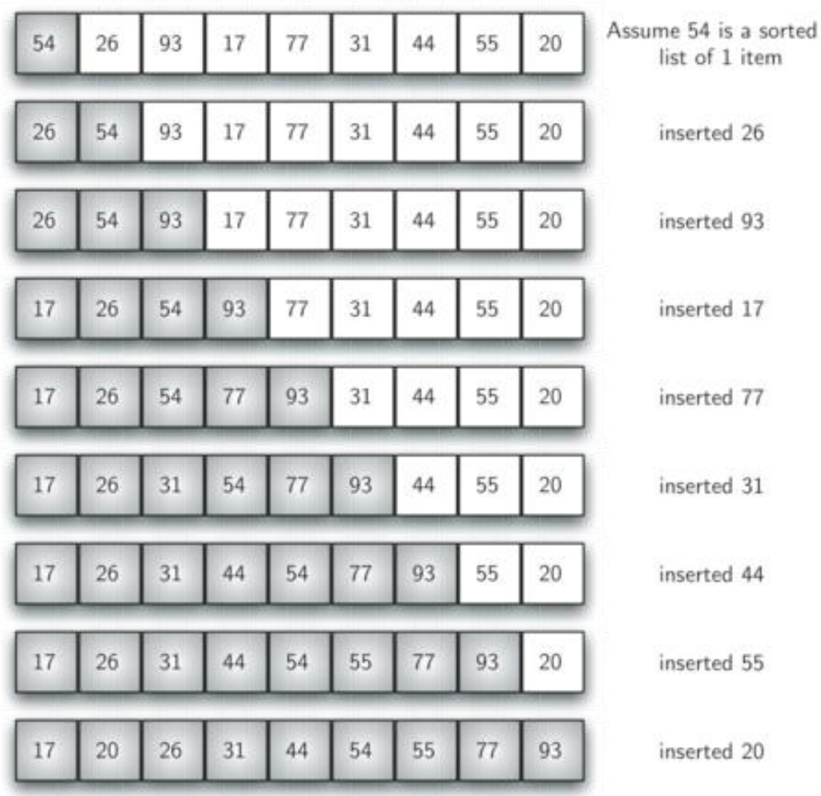
Selection Sort – the code in C

```
selection_sort(int s[], int n)
{
        int i,j;                        /* counters */
        int min;                        /* index of minimum */

        for (i=0; i<n; i++) {
                min=i;
                for (j=i+1; j<n; j++)
                        if (s[j] < s[min]) min=j;
                swap(&s[i],&s[min]);
        }
}
```

Insertion Sort

- Maintain two parts : sorted, unsorted
- Insertion : in every iteration i, pick i's element and insert it on the sorted part
- Two loops:
  - for-loop: i=1…..n-1
  - while-loop: j=i-1, …. 0 and
    i's value < j's value



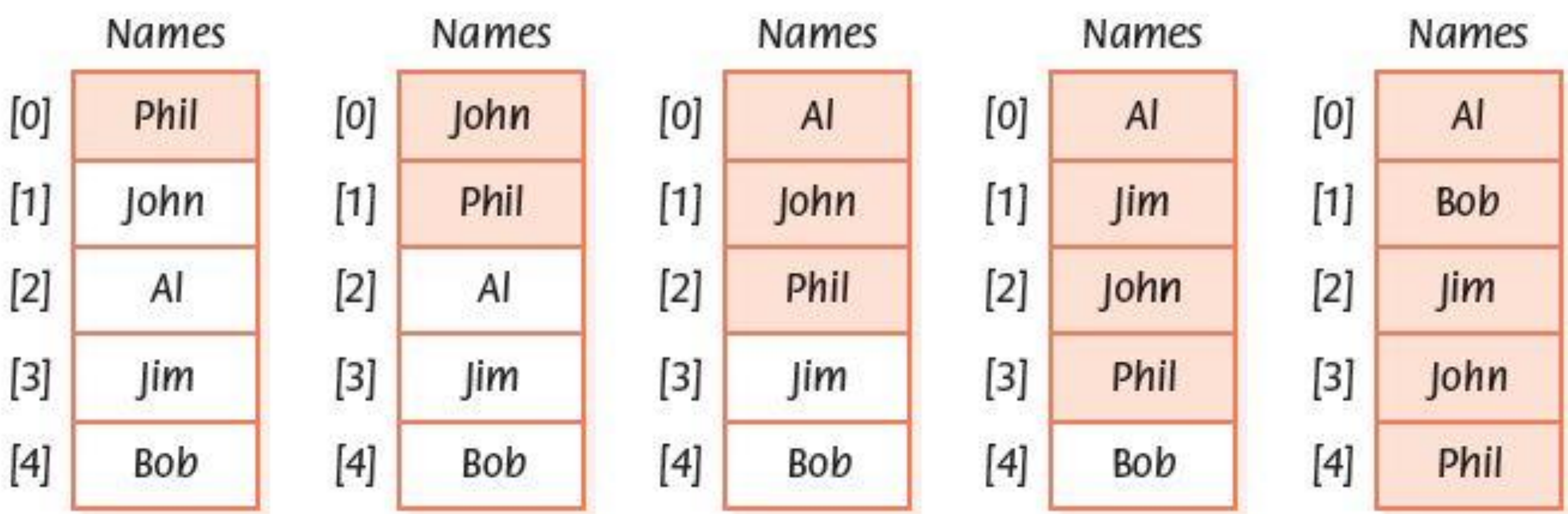


Insertion Sort

```
InsertionSort
Set current to 1
WHILE (current < length)
 Set index to current
 Set placeFound to FALSE
 WHILE (index > 0 AND NOT placeFound)
     IF (data[index] < data[index - 1])
          Swap data[index] and data[index - 1]
          Set index to index - 1
     ELSE
          Set placeFound to TRUE
 Set current to current + 1
```

Insertion Sort

- The item being added to the sorted portion can be bubbled up



Insertion Sort – the code in C

```
void insertionSort(int arr[], int n) {
  int i, key, j;
  for (i = 1; i < n; i++) {
    key = arr[i];
    j = i - 1;

    while (j >= 0 && arr[j] > key) {
      arr[j + 1] = arr[j];
      j = j - 1;
    }
    arr[j + 1] = key;
  }
}
```

Recursion

- **Some sorting algorithms leverage "recursion"**
- **Recursion :** the ability of a subprogram to call itself
- **Base case** : the case to which we have an answer
- **General case** : the case that expresses the solution in terms of a call to itself with a smaller version of the problem

- For example, the **factorial** of a number is defined as the number times the product of all the numbers between itself and 0:

$$N! = N * (N - 1)!$$

- Base case
  - Factorial(0) = 1 (0! is 1)
- General Case
  - Factorial(N) = N * Factorial(N-1)

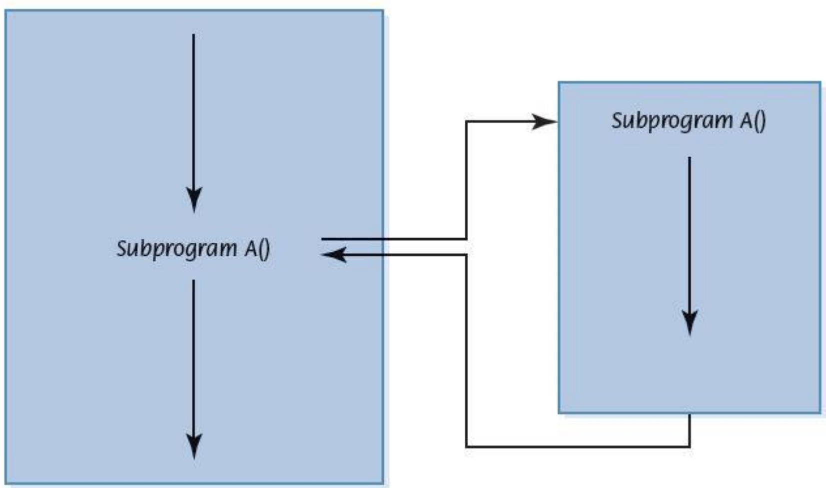
Subprogram Statements

- We can give a section of code a name and use that name as a statement in another part of the program
- When the name is encountered, the processing in the other part of the program halts while the named code is executed

- What if the subprogram needs data from the calling unit?
  - **Parameters** : Identifiers listed in parentheses beside the subprogram declaration; sometimes called **formal parameters**
  - **Arguments :** Identifiers listed in parentheses on the subprogram call; sometimes called **actual parameters**

Subprogram Statements

(a) Subprogram A does its task and calling unit continues with next statement



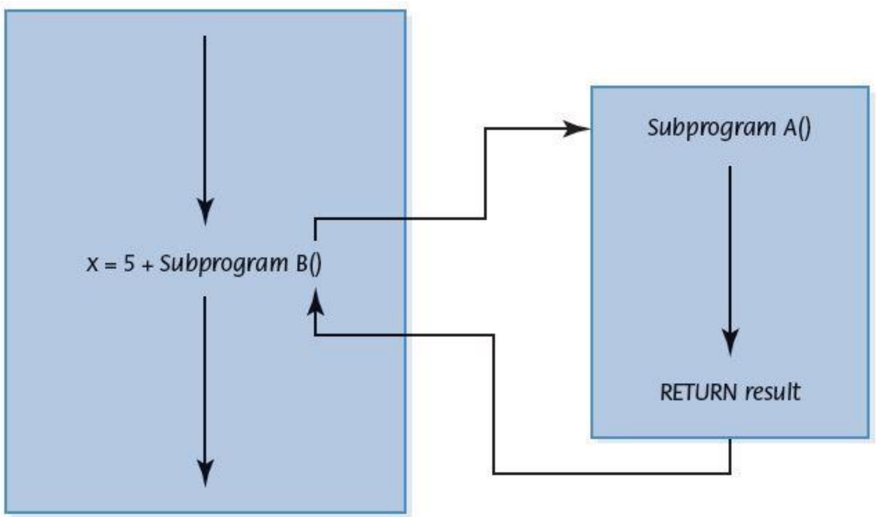


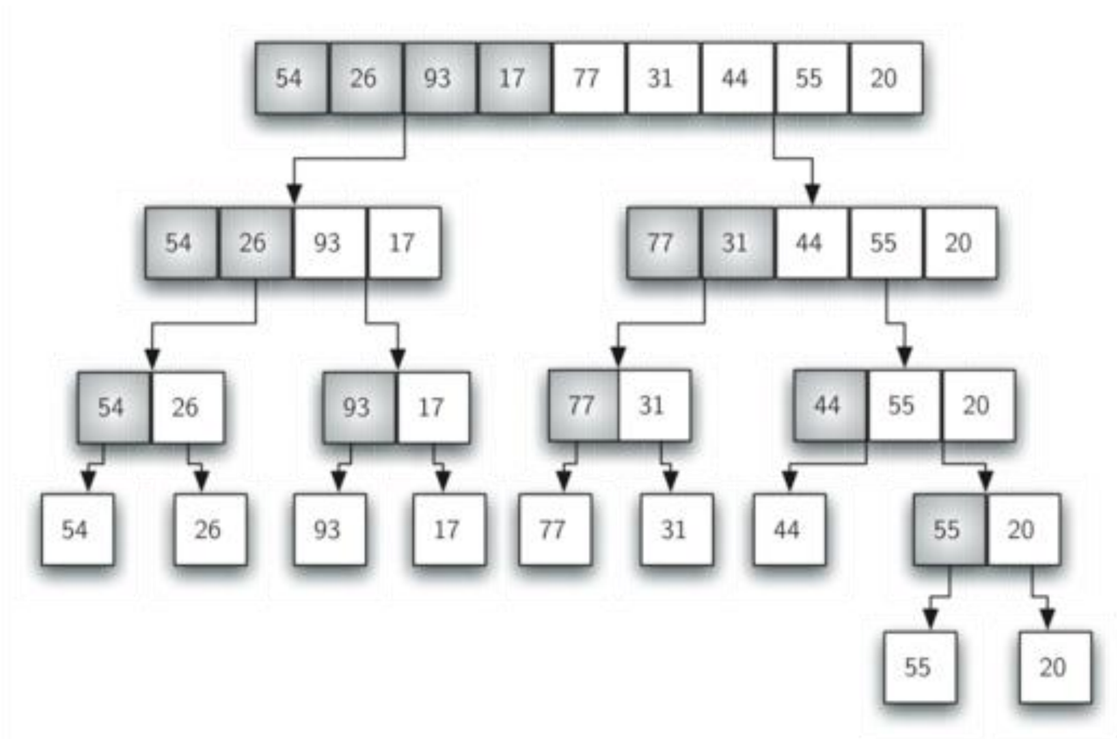
FIGURE 7.14 Subprogram flow of control

Recursion

```
BinarySearch (first, last)


IF (first > last)
      RETURN FALSE
ELSE
      Set middle to (first + last)/ 2
      IF (item equals data[middle])
                  RETURN TRUE
      ELSE
            IF (item < data[middle])
                  BinarySearch (first, middle - 1)
            ELSE
                  BinarySearch (middle + 1, last
```

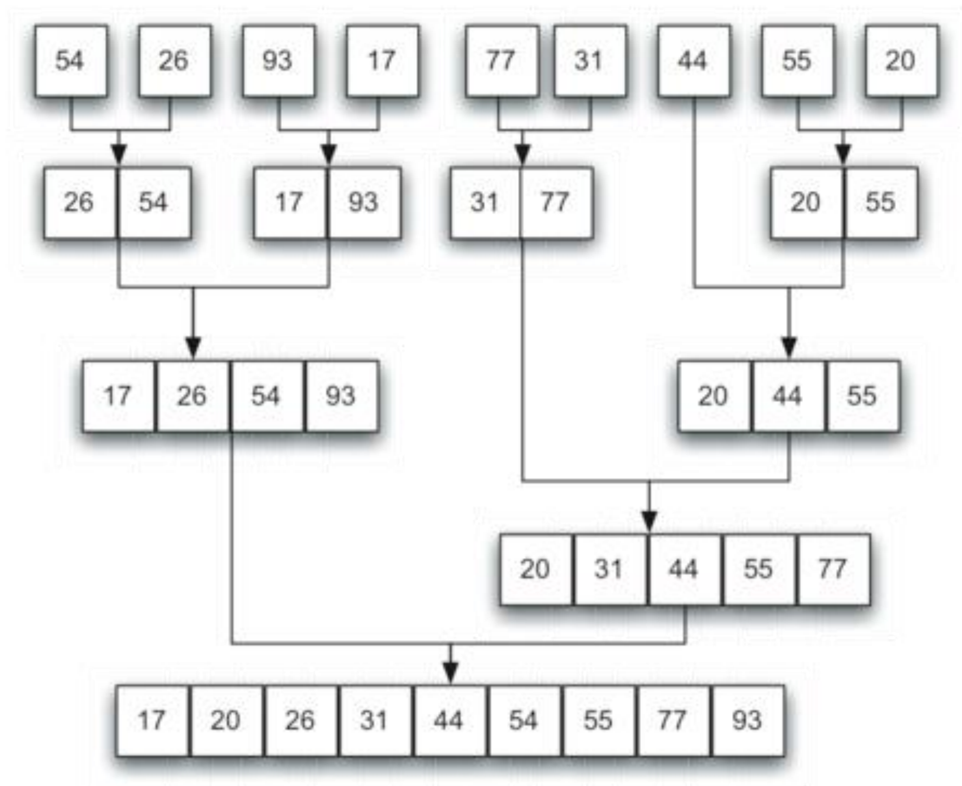
Merge sort – Divide & Conquer

- Divide



Merge sort – Divide & Conquer

- Conquer (merge)



Merge sort : divide process in Python

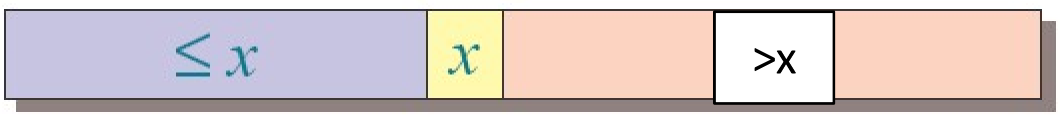
```
def mergeSort(alist):
    print("Splitting ",alist)
    if len(alist)>1:
        mid = len(alist)//2
        lefthalf = alist[:mid]
        righthalf = alist[mid:]

        mergeSort(lefthalf)
        mergeSort(righthalf)

         :
```

Merge process

```
def mergeSort(alist):
        if len(alist)>1:
        mid = len(alist)//2
        lefthalf = alist[:mid]
        righthalf = alist[mid:]
        mergeSort(lefthalf)
        mergeSort(righthalf)
        i=0
        j=0
        k=0
        while i < len(lefthalf) and j < len(righthalf):
            if lefthalf[i] < righthalf[j]:
                alist[k]=lefthalf[i]
                i=i+1
            else:
                alist[k]=righthalf[j]
                j=j+1
            k=k+1
        while i < len(lefthalf):
            alist[k]=lefthalf[i]
            i=i+1
            k=k+1
        while j < len(righthalf):
            alist[k]=righthalf[j]
            j=j+1
            k=k+1
```

Quick Sort

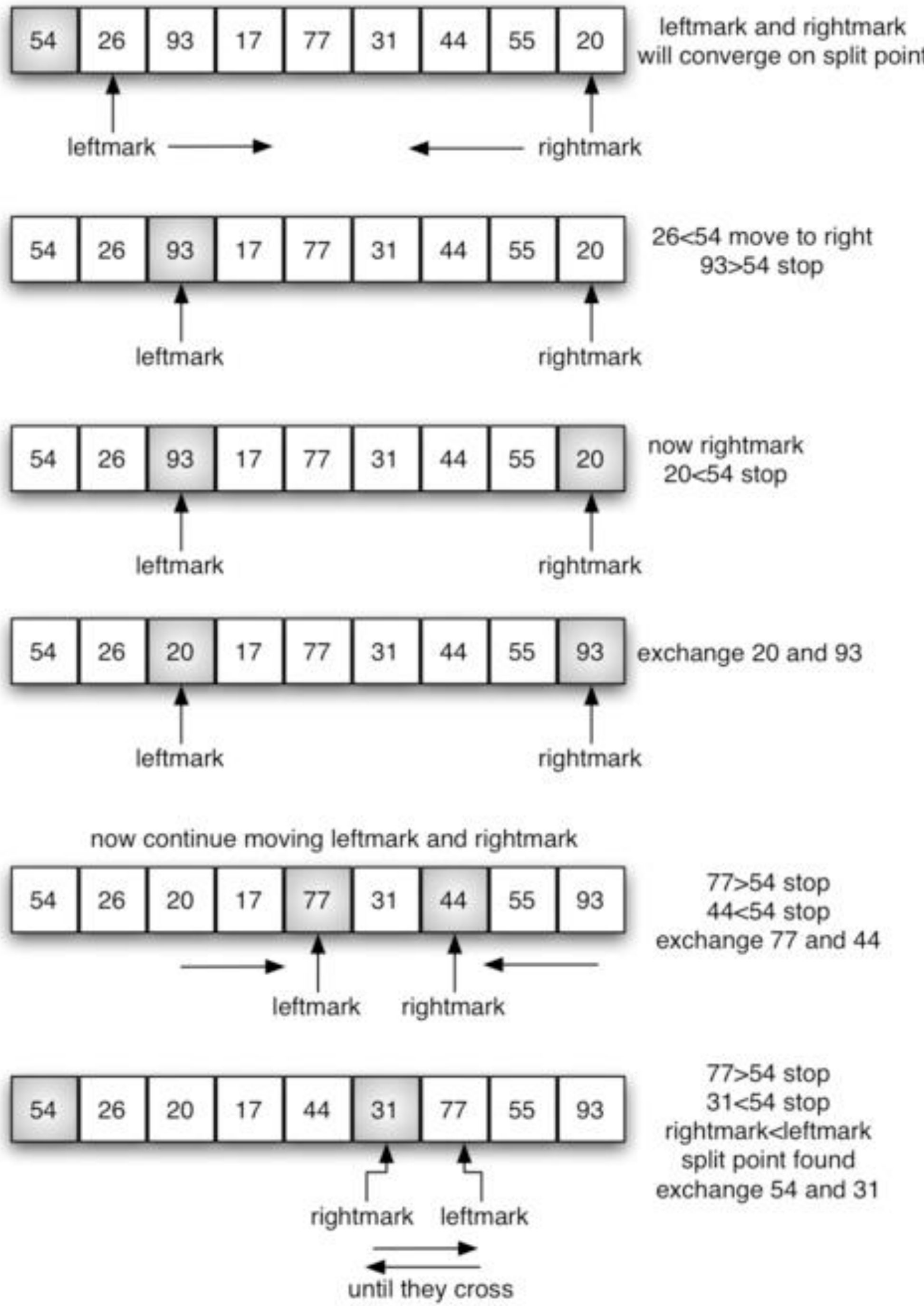
- Divide-and-Conquer
  - **Divide** the array into two parts
    - the value of x is called pivot



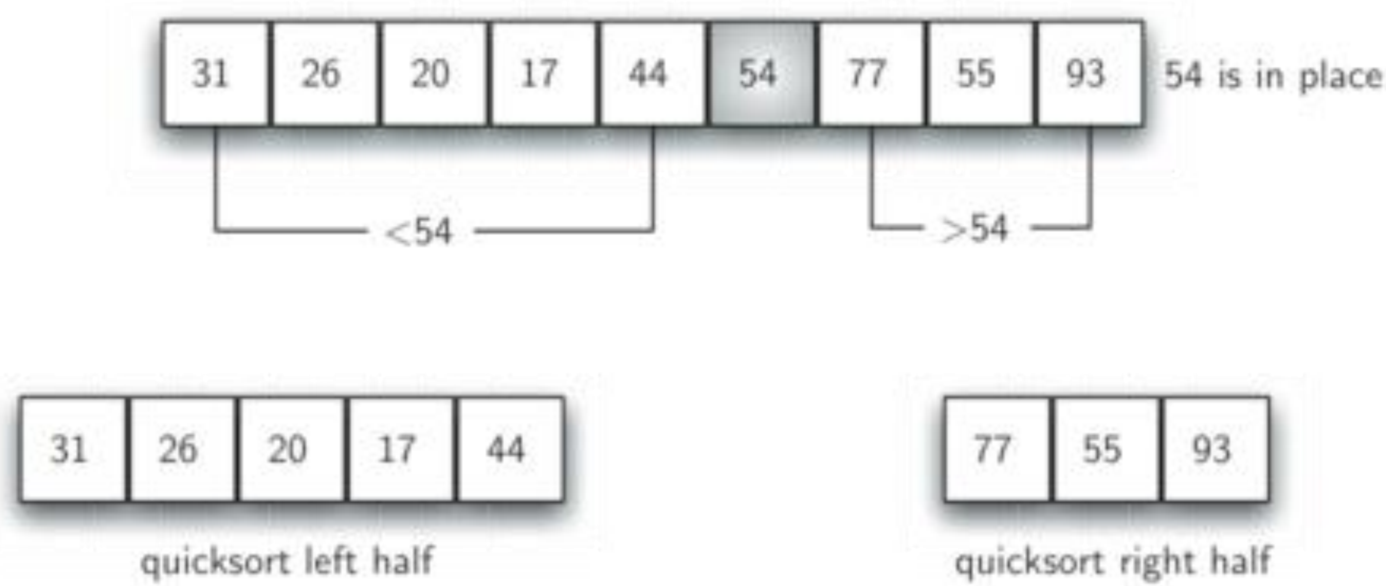
  - **Conquer**
    - do the same to each divided subarray
    - Combine

Quick Sort
- partition

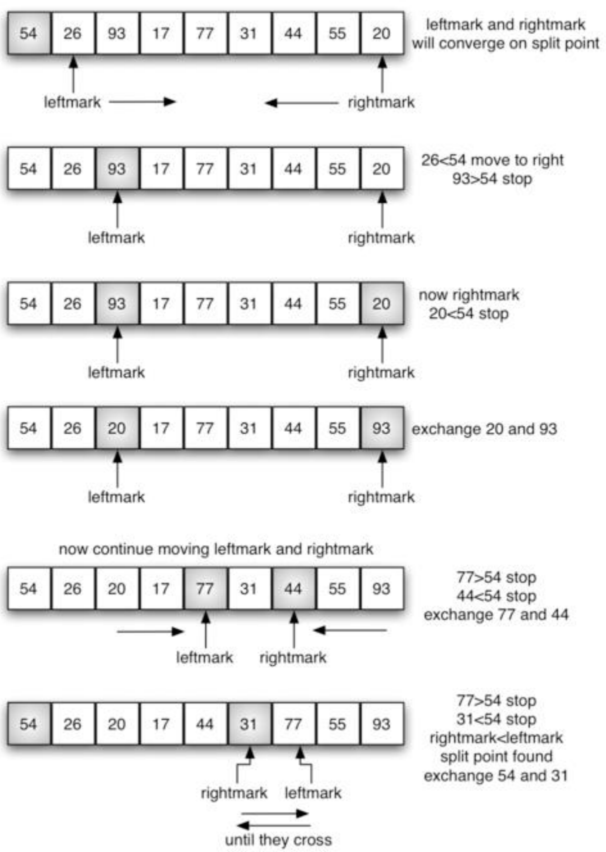
- pivot: 54



Quick Sort - partition



```
void quicksort(int a[], int l, int h)
{
    int p;
    if((h-l)>0) {
        p = partition(a, l, h);
        quicksort(a, l, p-1);
        quicksort(a, p+1, h);
    }
}
```




```
int partition(int a[], int first, int last) {
    int pivot, left, right;
    pivot = first;
    left = first;
    right = last;
    while (left < right) {
       while(a[left] <= a[pivot] && left < last)
         left++;
       while(a[right] > a[pivot] && right > first)
         right--;
       if(left < right) {
         Swap(&a[left], &a[right]);
       }
    }
    Swap(&a[pivot], &a[right]);
    return right;
}
```

Quick Sort

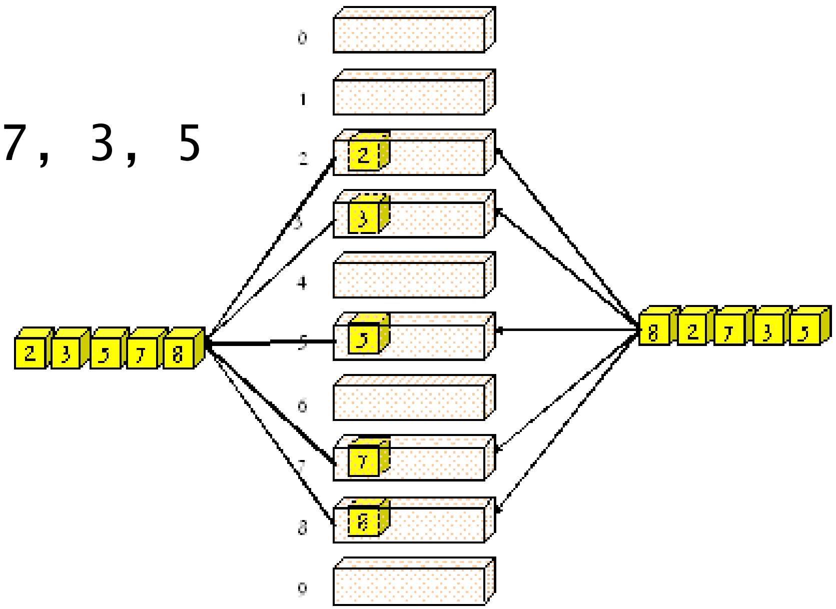
- With each attempt to sort an array of data elements, the array is divided at a splitting value, *pivot*, and the same approach is used to sort each of the smaller array (a smaller case)
  - This is very common approach of “divide and conquer”
- Process continues until the small arrays do not need to be divided further (= base case : the size is 1)
- The variables *first* and *last* in Quicksort algorithm reflect the part of the array *data* that is currently being processed

Radix Sort

- So far, we have seen comparison-based sorting algorithms
  - Their complexity : N^2 or N (Log N)

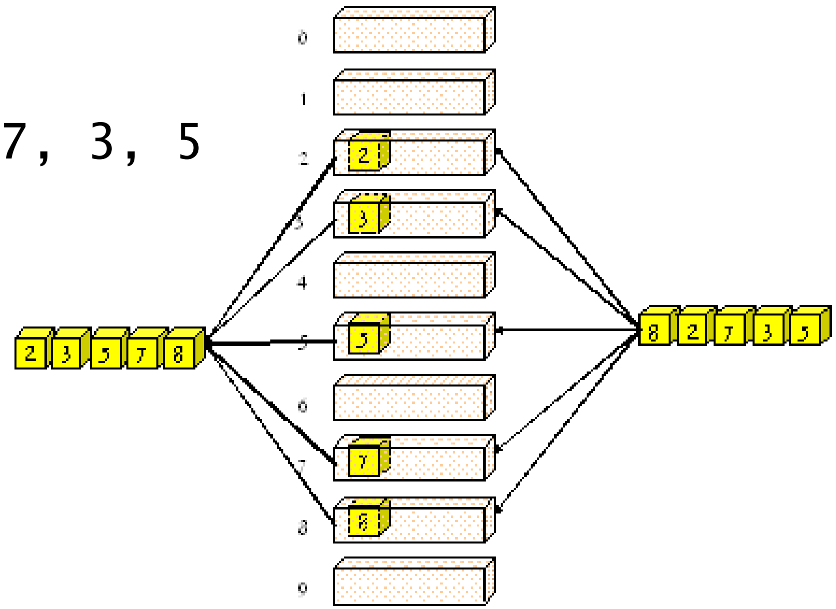
- What if we know the range of elements ? no comparison

8, 2, 7, 3, 5



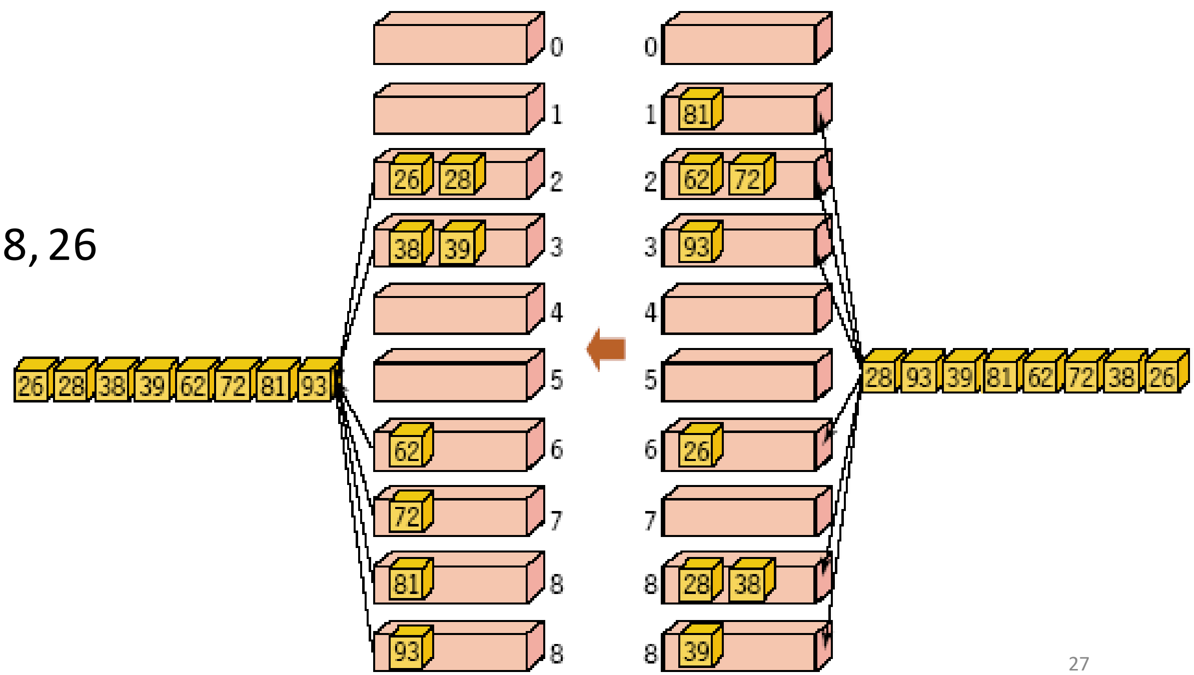
Radix Sort

8, 2, 7, 3, 5



- Two digits, more than …

28, 93, 39, 81, 62, 72, 38, 26



Quiz : Vito's Family

The input consists of several test cases. The first line contains the number of test cases.
For each test case you will be given the integer number of relatives r $(0 < r < 500)$ and the street numbers (also integers) $s_1, s_2, \ldots, s_i, \ldots, s_r$ where they live $(0 < s_i < 30,000)$. Note that several relatives might live at the same street number.

For each test case, your program must write the minimal sum of distances from the optimal Vito's house to each one of his relatives. The distance between two street numbers s_i and s_j is $d_{ij} = |s_i - s_j|$.

Sample Input

```
2
2 2 4
3 2 4 6
```

Sample Output

```
2
4
```

Vito’s Family

- 2, 4
  - If v = 2
  - If v = 4

- 2, 4, 6
  - If v=2
  - If v=4
  - If v=6

Discussion : radix sort

- How to sort three-digit integers using radix sort algorithm ?