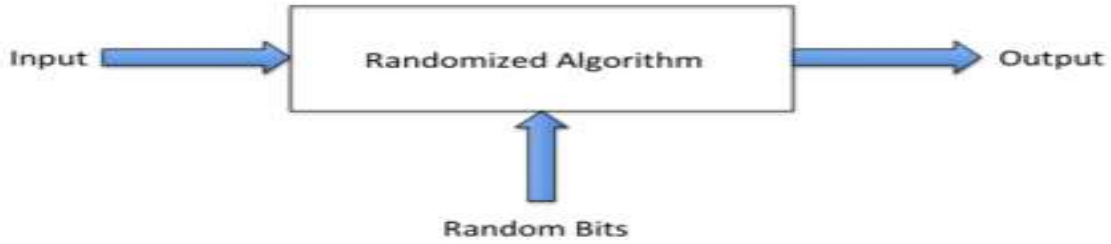
Randomized Algorithms

Deterministic Algorithms

- Goal: To solve a computational problem correctly and efficiently.
- Behaviour of the algorithm is determined completely by the input.
- Upon reruns, the algorithm executes in exactly the same manner.

- Predefined set of inputs and required output that follow predefined algorithm steps
- We define these algorithms and design these algorithms that for this particular specified input, this is the desired output.

Randomized Algorithms



- In addition to the input, the algorithm execution depends on some random bits as well.
- Behaviour of the algorithm is not determined completely by the input.
- Upon reruns, the algorithm can execute in a different manner, even with the same input.

- In the algorithm part, we introduce something known as random bits.
- So, this gets added to your input and change the nature of the required output.
- A **randomized algorithm** is an algorithm which employs a degree of randomness as part of its logic.

Randomized Algorithms

- An algorithm that uses random numbers to decide what to do next anywhere in its logic is called Randomized Algorithm.
- For example, in Randomized Quick Sort, we use random number to pick the next pivot (or we randomly shuffle the array).
- Typically, this randomness is used to reduce time complexity or space complexity in other standard algorithms.

Advantages and Disadvantages

Advantages

- Simplicity
- In some cases, faster than deterministic
- In some cases, no deterministic algorithm exists.

Disadvantages

- Randomness is a resource.
- With some probability, we can get an incorrect output.
- With some probability, can perform in worst case time.

Types

Las Vegas Algorithms

- Correctness is guaranteed
- May not be fast always.
- Probability of worst case running time is small.
- Expected running time $<$ Worst case running time

Monte Carlo Algorithms

- Running time is fixed.
- Correctness of the algorithm need not be assured.
- Probability of an incorrect output is small.

Sorting

- Given the sequence of ‘n’ nos a1, a2,…an.
- Reorder the input sequence such that

  $a1 \leq a2 \leq a3 \leq \ldots. \leq an$

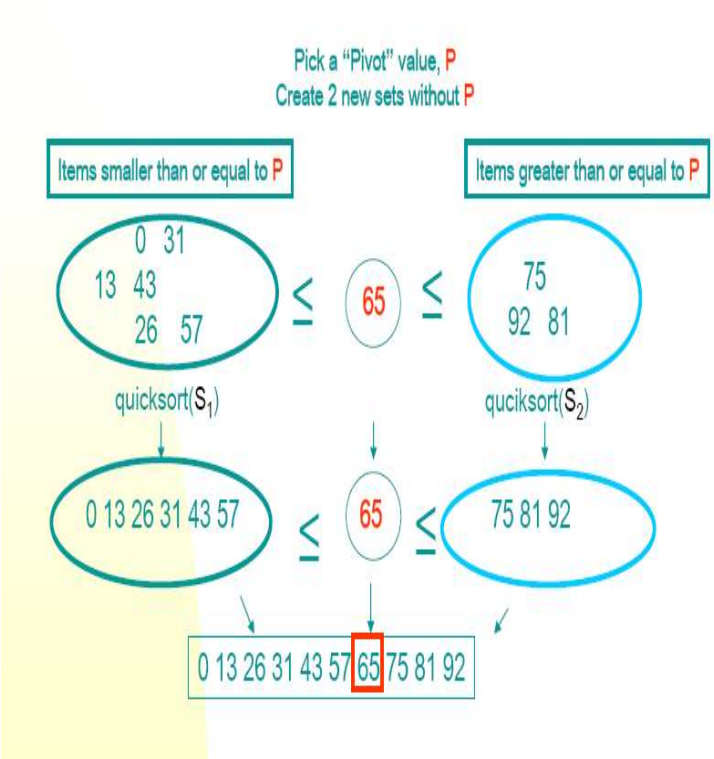
Quick Sort

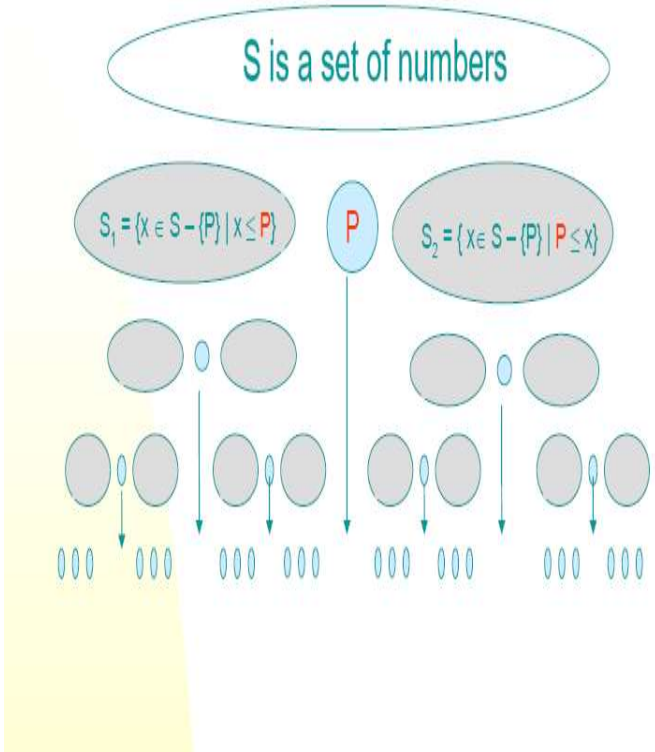
- **Quick sort** or **partition-exchange sort**, is a sorting algorithm developed by Tony Hoare.
- Quick sort is a Comparison sort.
- Quick sort is a divide and conquer algorithm (divides the array into sub arrays).
- As the name implies, it is quick and it is the algorithm generally preferred for sorting.

Basic Idea

- Pick an element, say ***P*** (the pivot)
- Re-arrange the elements into 3 sub-blocks,
  1. those less than or equal to (≤) ***P*** (the **left-**block S_1)
  2. ***P*** (the only element in the **middle-**block)
  3. those greater than or equal to (≥) ***P*** (the **right-**block S_2)
- Repeat the process **recursively** for the **left-** and **right-** sub-blocks. Return {quicksort(S_1), ***P***, quicksort(S_2)}. (That is the results of quicksort(S_1), followed by ***P***, followed by the results of quicksort(S_2))

Basic Idea


Pick a "Pivot" value, P
Create 2 new sets without P
Items smaller than or equal to P
Items greater than or equal to P
0 31
13 43
26 57
≤
65
≤
75
92 81
quicksort(S1)
quciksort(S2)
0 13 26 31 43 57
≤
65
≤
75 81 92
0 13 26 31 43 57 65 75 81 92


S is a set of numbers
S1 = {x ∈ S – {P} | x ≤ P}
P
S2 = {x ∈ S – {P} | P ≤ x}

Basic Idea

Note:

- The main idea is to find the "right" position for the pivot element ***P***.
- After each **"pass"**, the pivot element, ***P***, should be **"in place".**
- Eventually, the elements are sorted since each pass puts at least one element (i.e., ***P***) into its final position.

Issues:

- How to choose the pivot ***P*** ?
- How to partition the block into sub-blocks?

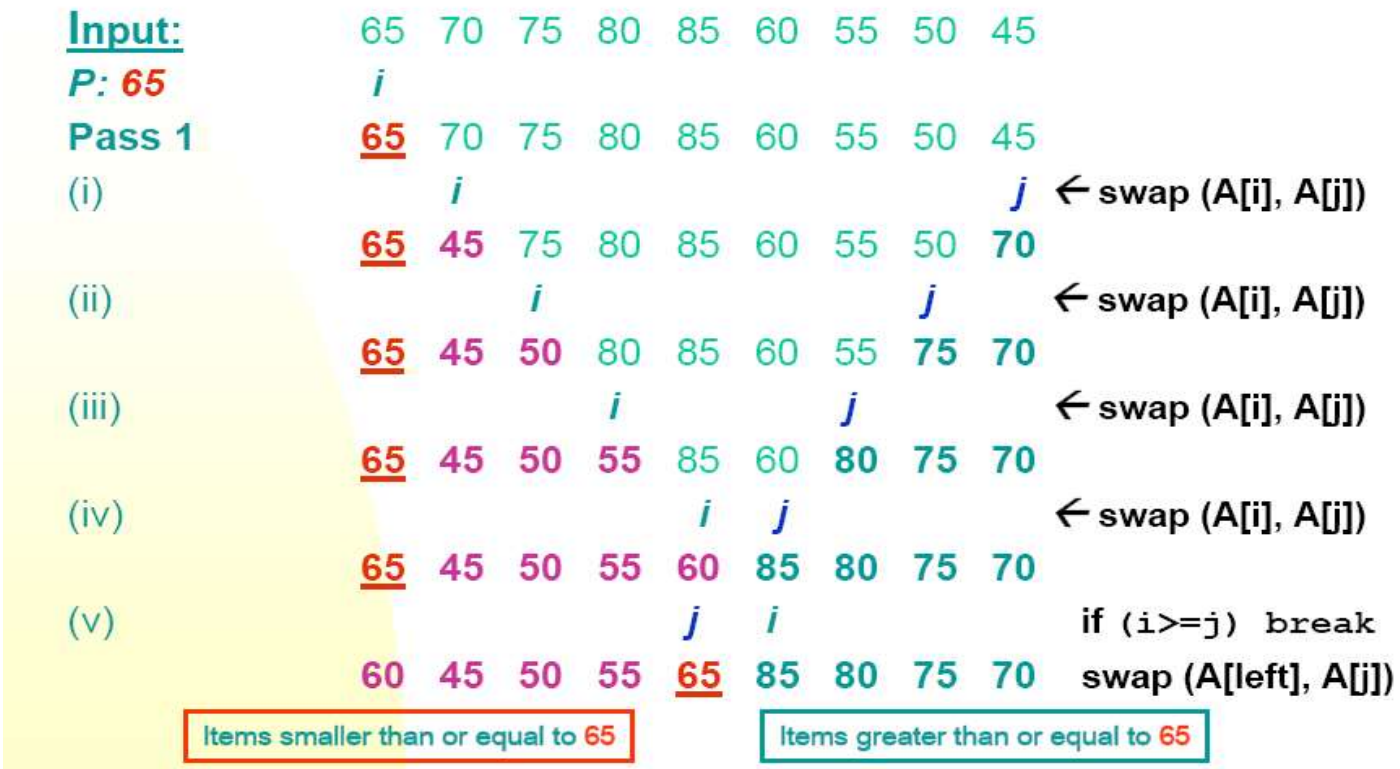
Pivot Selection

- Generally we choose an array element in the first position to use as the pivot

Other option:

1. Choosing last element
2. Choosing middle element

Partitioning Example



Example

Result of Pass 1: 3 sub-blocks:

60 45 50 55 **65** 85 80 75 70

Pass 2a (left sub-block): 60 45 50 55 (P = 60)

i j

60 45 50 55

j i if (i>=j) break

55 45 50 60 swap (A[left], A[j])

Pass 2b (right sub-block): 85 80 75 70 (P = 85)

i j

85 80 75 70

j i if (i>=j) break

70 80 75 85 swap (A[left], A[j])

```
Quicksort (K,LB,UB)
        flag = true
        if LB < UB then
                i = LB
                j = UB + 1
                key = K[i]
                repeat while flag
                        i = i+1
                        repeat while K[i] < key
                                i = i+1
                        j = j-1
                        repeat while K[j] > key
                                j = j-1
                        if  i < j then swap K[i] <-> K[j] else flag = false
                swap key <-> K[j]
                call Quicksort (K,LB,j-1)
                call Quicksort (K,j+1,UB)
```

Time complexity

- The worst case time complexity of a typical implementation of QuickSort is $O(n^2)$.
- The worst case occurs when the picked pivot is always an extreme (smallest or largest) element. This happens when input array is sorted or reverse sorted and either first or last element is picked as pivot.

- Average case complexity = O(n log n)

Randomized Quicksort

- To get better performance on sorted or nearly sorted data, we can randomize the algorithm to get the same effect as if the input data were random.
- Instead of explicitly permuting the input data (which is expensive), randomization can be accomplished trivially by **random sampling** of one of the array elements as the pivot.

```
Quicksort (K,LB,UB)
        flag = true
        if LB < UB then
                x = random(LB, UB)
                exchange K[x] with K[LB]
                i = LB
                j = UB + 1
                key = K[i]
                repeat while flag
                        i = i+1
                        repeat while K[i] < key
                                i = i+1
                        j = j-1
                        repeat while K[j] > key
                                j = j-1
                        if  i < j then swap K[i] <-> K[j] else flag = false
                swap key <-> K[j]
                call Quicksort (K,LB,j-1)
                call Quicksort (K,j+1,UB)
```