Divide and Conquer

Divide and Conquer

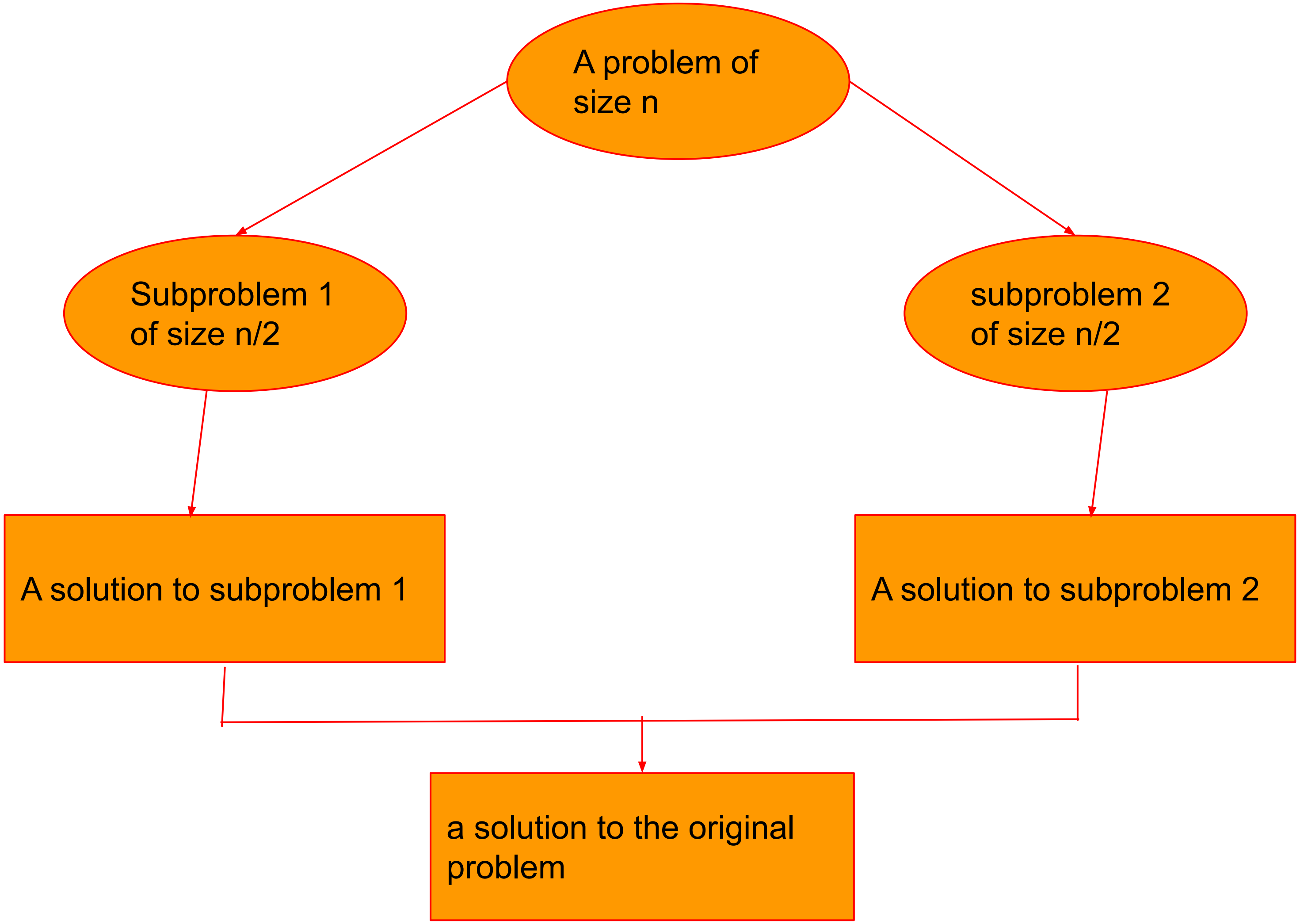
- **Divide** the problem into a number of subproblems
- **Conquer** the subproblems (solve them)
- **Combine** the subproblem solutions to get the solution to the original problem
-
- Note: often the "conquer" step is done recursively

Divide and Conquer

- A general methodology for using recursion to design efficiently algorithms

- It solves a problem by:
  - Dividing the data into parts
  - Finding sub solutions for each of the parts
  - Constructing the final answer from the sub solutions

Divide and Conquer



Divide and Conquer Algo.

- **Divide** the problem into a number of subproblems
  - Subproblems must of same type
  - Subproblems do not need to overlap

- **Conquer** by solving sub-problems recursively. If the sub-problems are small enough, solve them in brute force fashion
- **Combine** the solutions of sub-problems into a solution of the original problem

Divide and Conquer

- For divide-and-conquer algorithms the running time is mainly affected by 3 criteria:
  - The number of sub-instance into which a problem is split.
  - The ratio of initial problem size to sub-problem size.
  - The number of steps required to divide the initial instance and to combine sub solutions.

Analyzing Divide and Conquer Algo.

- When an algorithm contains a recursive call to itself, its running time can often be described by a recurrence equation which describes the overall running time on a problem of size n in terms of the running time on smaller inputs.
- For divide-and-conquer algorithms, we get recurrences that looks like:

$$T(n) = \begin{cases} \Theta(1) & \text{if } n \leq c \\ aT(n/b) + D(n) + C(n) & \text{otherwise} \end{cases}$$

$$T(n) = \begin{cases} \Theta(1) & \text{if } n \leq c \\ aT(n/b) + D(n) + C(n) & \text{otherwise} \end{cases}$$

- where a=the number of subproblems we break the problem into
- n/b=the size of the subproblems (in terms of n)
- D(n) is the time to divide the problem of size n into the subproblems
- C(n) is the time to combine the subproblem solutions to get the answer for the problem of size n

Example: Divide and Conquer

- Binary Search
- Heap Construction
- Tower of Hanoi
- Exponentiation
  - Fibonacci Sequence
- Quick sort
- Merge Sort
- Multiplying large integers
- Matrix Multiplication
- Closest Pairs

Merge Sort

Merge Sort

Recursive in structure

- Divide the problem into subproblems that are similar to the original but smaller in size
- Conquer the subproblems by solving them recursively. If they are small enough, just solve them in a straightforward manner.
- **Combine** the solutions to create a solution to the original problem

An Example: Merge Sort

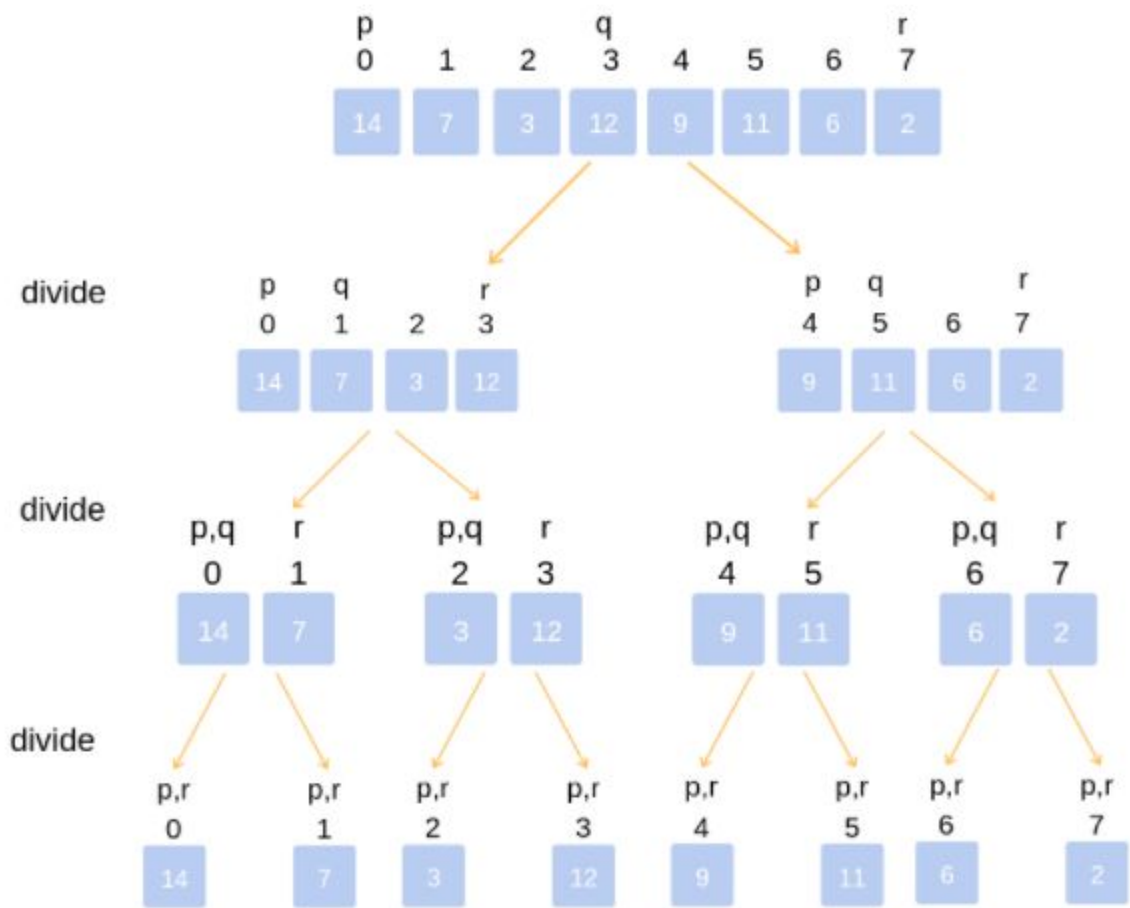
Sorting Problem: Sort a sequence of n element into non-decreasing order.

Divide: Divide the n-element sequence to be sorted into two subsequences of n/2 elements each

Conquer: Sort the two subsequences recursively using merge sort

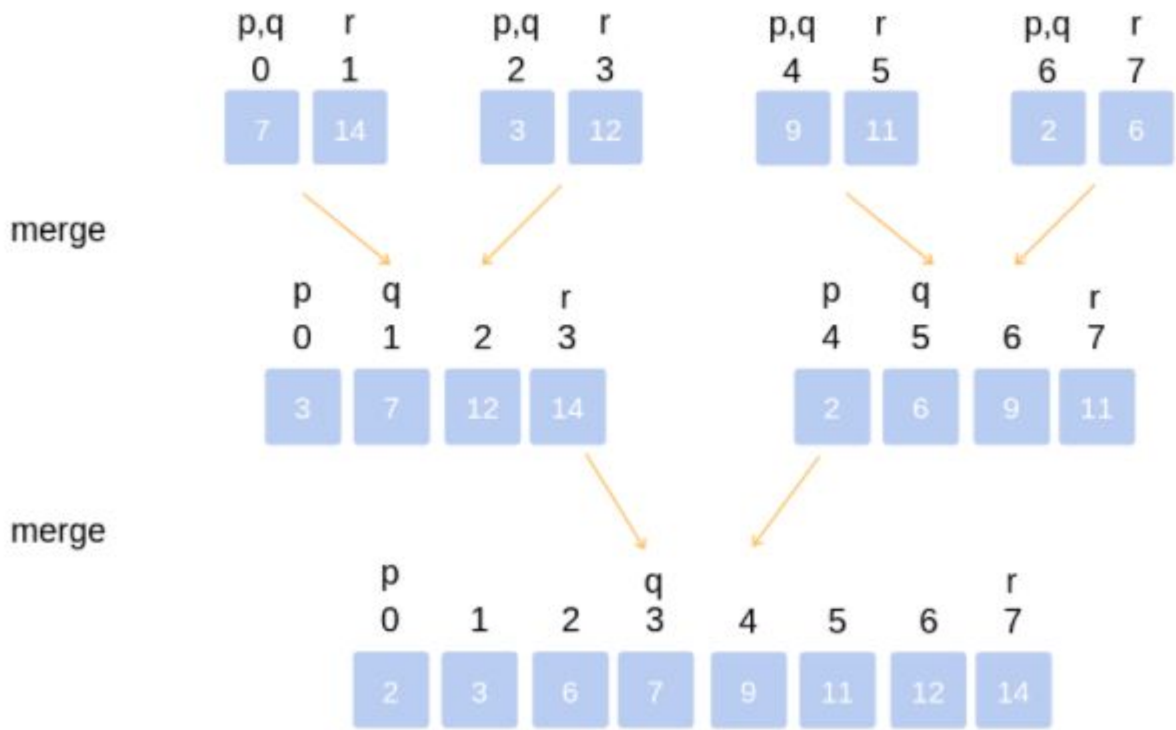
Combine: Merge the two sorted subsequences to produce the sorted.

An Example: Merge Sort



Top-down Implementation

An Example: Merge Sort



Merging of two lists

Implementation Of Merge Sort

```
void mergeSort(int *Arr, int start, int end) {

        if(start < end) {
                int mid = (start + end) / 2;
                mergeSort(Arr, start, mid);
                mergeSort(Arr, mid+1, end);
                merge(Arr, start, mid, end);
        }
}
```

Implementation Of Merge Sort

```
void merge(int *Arr, int start, int mid, int end) {
	// create a temp array
	int temp[end - start + 1];
	// crawlers for both intervals and for temp
	int i = start, j = mid+1, k = 0;
	// traverse both arrays and in each iteration add smaller of both elements in temp
	while(i <= mid && j <= end) {
		if(Arr[i] <= Arr[j]) {
			temp[k] = Arr[i];
			k += 1; i += 1;
		}
		else {
			temp[k] = Arr[j];
			k += 1; j += 1;
		}
	}
	// add elements left in the first interval
	while(i <= mid) {
		temp[k] = Arr[i];
		k += 1; i += 1;}
	// add elements left in the second interval
	while(j <= end) {
		temp[k] = Arr[j];
		k += 1; j += 1; }
	// copy temp to original interval
	for(i = start; i <= end; i += 1) {
		Arr[i] = temp[i - start]}
}
```

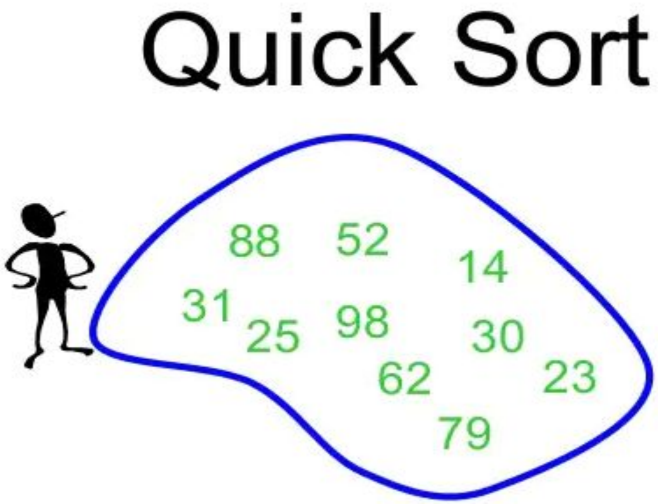
Merge Sort

how does merge sort stack up?

time complexity	O(n log n)
space complexity	out-of-place
stability	stable
internal/external?	external
recursive/non-recursive?	recursive
comparison sort?	comparison

QuickSort

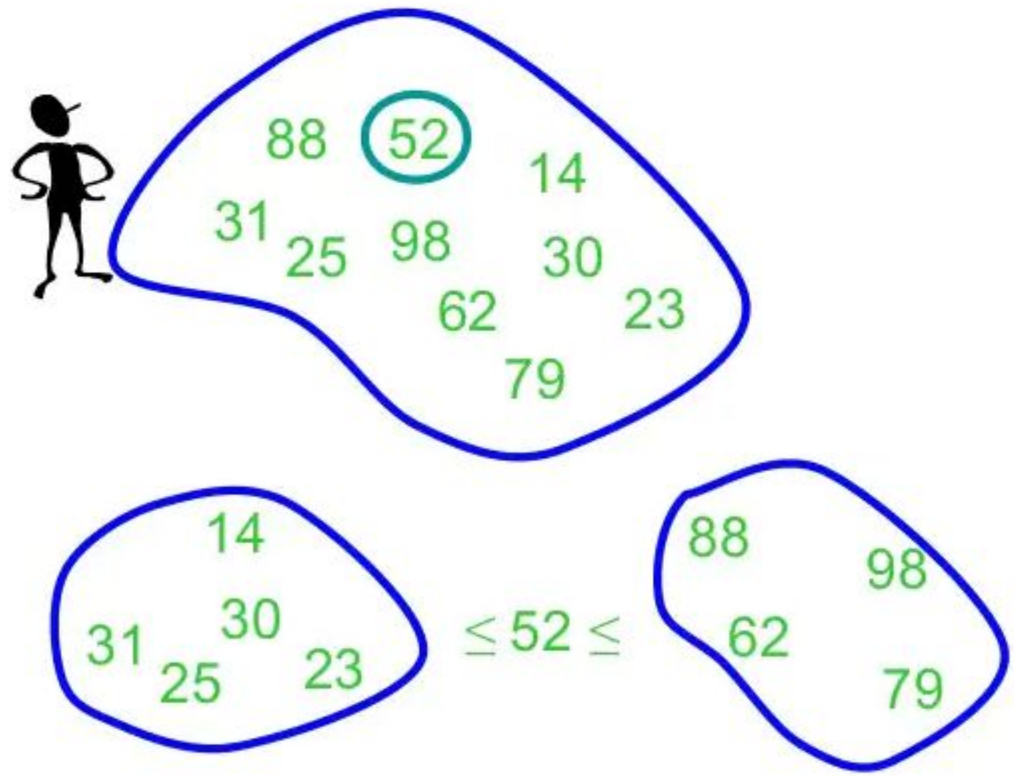
Quick Sort



Divide and Conquer

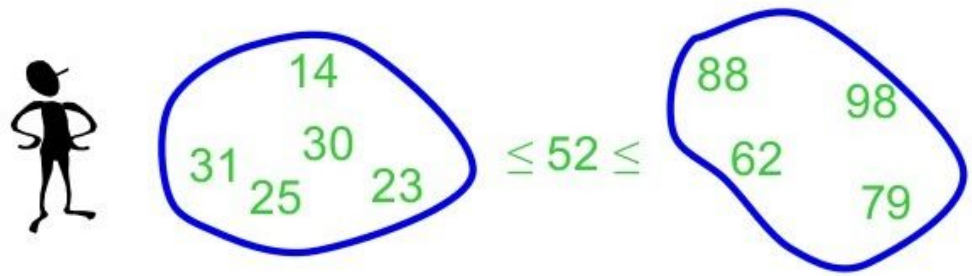
Quick Sort

Partition set into two using randomly chosen pivot

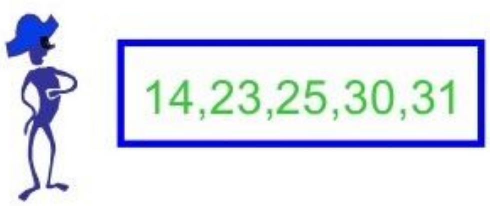


Quick Sort

Quick Sort



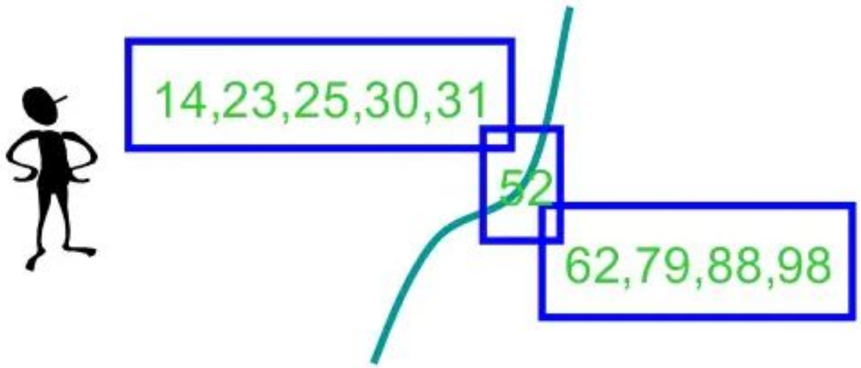
sort the first half.



sort the second half.

Quick Sort

Quick Sort



Glue pieces together.

14,23,25,30,31,52,62,79,88,98

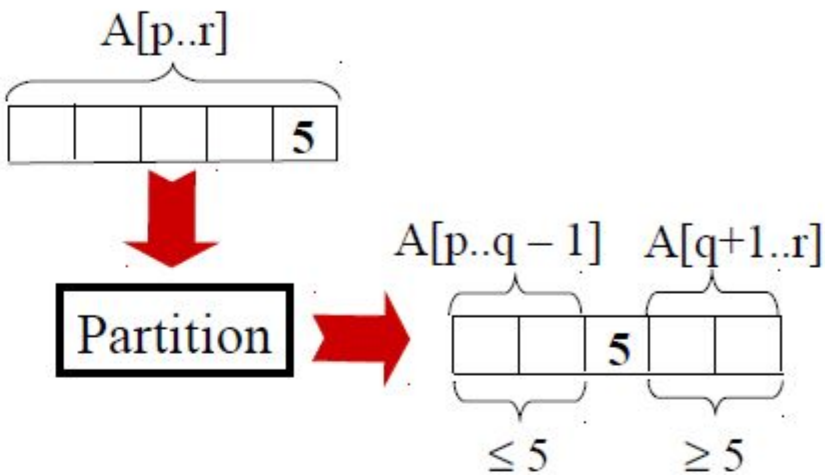
Quick Sort: Design

- Follows the **divide-and-conquer** paradigm
- **Divide:** Partition (separate) the array A[p..r] into two (possibly nonempty) subarrays A[p..q-1] and A[q+1..r].
  - Each element in A[p..q-1]<=A[q]
  - A[q]<= each element in A[q+1,r]
  - Index q is computed as part of the partitioning procedure.
- **Conquer:** Sort the two subarrays A[p..q-1]& A[q+1..r]by recursive calls to quicksort.
- **Combine:** Since the subarrays are sorted in place-no work is needed to combine them.

Quick Sort: Pseudocode

```
Quicksort(A, p, r)
    if p < r then
        q := Partition(A, p, r);
        Quicksort(A, p, q – 1);
        Quicksort(A, q + 1, r)
    fi
```

```
Partition(A, p, r)
    x:= A[r],
    i :=p – 1;
    for j := p to r – 1 do
        if A[j] ≤ x then
            i := i + 1;
            A[i] ↔ A[j]
        fi
    od;
    A[i + 1] ↔ A[r];
    return i + 1
```



Quick Sort: Example

	p	r
initially:	2 5 8 3 9 4 1 7 10 **6**	**note:** pivot (x) = 6
	i j	
next iteration:	2 5 8 3 9 4 1 7 10 **6**	
	i j	
next iteration:	2 5 8 3 9 4 1 7 10 **6**	
	i j	
next iteration:	2 5 8 3 9 4 1 7 10 **6**	
	i j	
next iteration:	2 5 3 8 9 4 1 7 10 **6**	
	i j	

```
Partition(A, p, r)
    x, i := A[r], p – 1;
    for j := p to r – 1 do
        if A[j] ≤ x then
            i := i + 1;
            A[i] ↔ A[j]
        fi
    od;
    A[i + 1] ↔ A[r];
    return i + 1
```

Quick Sort: Example

next iteration:	2	5	3	8	9	4	1	7	10	**6**
			i		j					
next iteration:	2	5	3	8	9	4	1	7	10	**6**
			i			j				
next iteration:	2	5	3	4	9	8	1	7	10	**6**
				i			j			
next iteration:	2	5	3	4	1	8	9	7	10	**6**
					i			j		
next iteration:	2	5	3	4	1	8	9	7	10	**6**
					i				j	
next iteration:	2	5	3	4	1	8	9	7	10	**6**
					i					j
after final swap:	2	5	3	4	1	**6**	9	7	10	8
					i					j

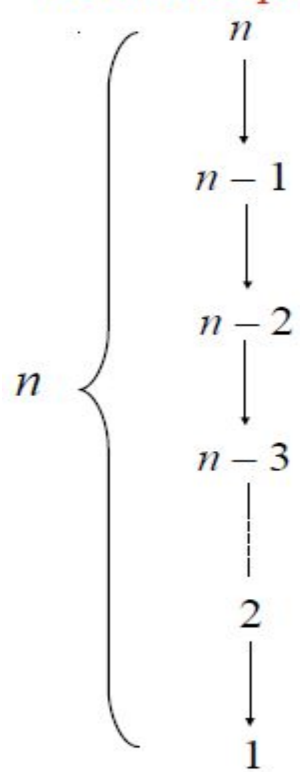
```
Partition(A, p, r)
    x, i := A[r], p – 1;
    for j := p to r – 1 do
        if A[j] ≤ x then
            i := i + 1;
            A[i] ↔ A[j]
        fi
    od;
    A[i + 1] ↔ A[r];
    return i + 1
```

Algorithm Performance

- Running time of quicksort depends on whether the partitioning is balanced or not.
- Worst-case partitioning(unbalanced partitions):
  - Occurs when every call to partition results in the most unbalanced partition.
  - Partition is most unbalanced when
    - Subproblem 1 is of size n-1, and subproblems 2 is of size 0 or vice versa.
    - pivot>=every element in A[p..r-1] or pivot < every element in A[p..r-1]
  - Every call to partition is most unbalanced when
    - Array A[1..n] is sorted or reverse sorted.

Worst-case Partition Anaysis

Recursion tree for worst-case partition



- Running time for worst-case partitions at each recursive level:
- $T(n) = T(n – 1) + T(0) +$ PartitionTime(n)
- $= T(n – 1) + \Theta(n)$
- $= \sum_{k=1 \text{ to } n} \Theta(k)$
- $= \Theta(\sum_{k=1 \text{ to } n} k)$
- $= \Theta(n^2)$

Best-case Partitioning

- Size of each subproblem <=n/2.
  - one of the subproblems is of size n/2
  - the other is of size n/2-1
- Recurrence for running time
  - $T(n) <= 2T(n/2) + \text{Partition Time}(n)$

$$= 2T(n/2) + \Theta(n)$$

- **T(n)=Θ(n lg n)**

Randomized QuickSort

Randomized QuickSort

- An algorithm is randomized if its behavior is determined not only by the input but also by values produced by a random-number generator.
- Exchange A[r] with an element chosen at random from A[p…r] in Partition.
- This ensures that the pivot element is equally likely to be any of input elements
- We can sometimes add randomization to an algorithm in order to obtain good average-case performance over all inputs.

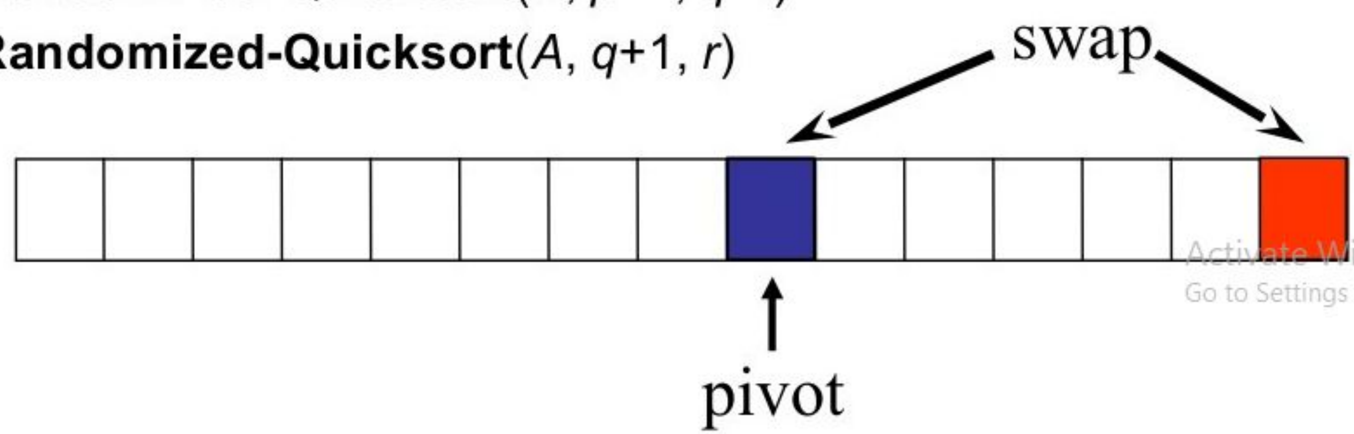
Randomized QuickSort

Randomized-Partition(*A, p, r*)

1. $i \leftarrow Random(p, r)$
2. exchange $A[r] \leftrightarrow A[i]$
3. **return Partition**(A, p, r)

Randomized-Quicksort(*A, p, r*)

1. **if** $p < r$
2.   **then** $q \leftarrow$ **Randomized-Partition**(A, p, r)
3.   **Randomized-Quicksort**$(A, p, q\text{-}1)$
4.   **Randomized-Quicksort**$(A, q\text{+}1, r)$



QuickSort

Time Complexity	
Best	O(n*log n)
Worst	$O(n^2)$
Average	O(n*log n)
Space Complexity	O(log n)
Stability	No