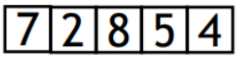
Sorting: Bubble sort, Selection sort, Quick sort,

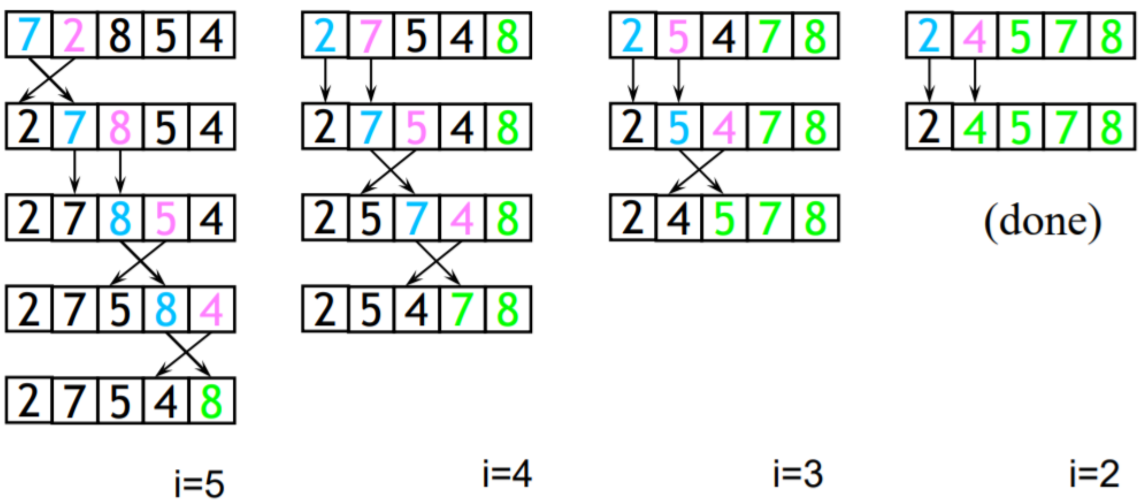
Bubble sort, selection sort and quick sort.

Bubble sort

- Popular but inefficient.
- Works by swapping elements:
  - The heaviest bubble goes to the bottom.
  - Or the lightest bubble goes to the top.
- In-place sorting. Uses on temp value.
- Complexity: $\Theta(n^2)$



- BubbleSort(A[1..n]: int)
  - **for** i = n **downto** 1 **do**
    - **for** j =2 **to** i **do**
      - **if** A[j-1] > A[j] **then** swap(A, j-1, j)



Intertion sort

- **In-place** sorting.
  - Only a **constant** number of elements of the input array are ever stored outside the array.
- Worst case complexity: $\Theta(n^2)$

Merge sort

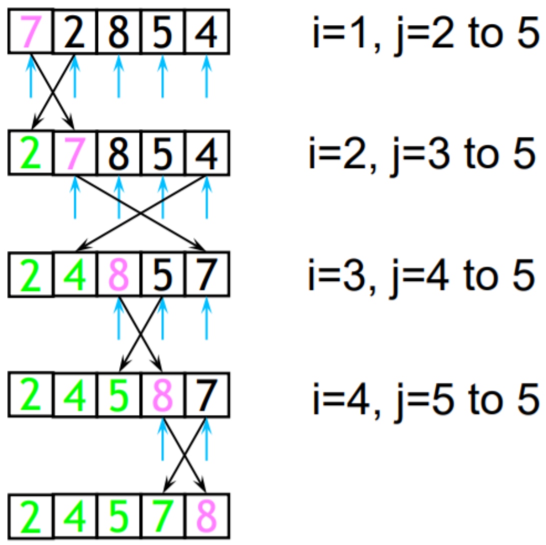
- Uses divide-and-conquer technique.
- Worst case complexity: $\Theta(n * lg(n))$ (Better than interstion sort)

- Not in-place sorting.
  - Merge stop: Uses extra memory for sorting

Selection Sort

- Search elements *i* trough *n* and select the smallest number
  - Swap it with the element in location *i*.
- Continue until nothing left to search.

```
for i= 1 to n-1 do
    min=i;
    for j=i+1 to n do
        if A[j] < A[min] then min=j ;
    temp = A[i];
    A[i] = A[min];        swap(A, i, min)
    A[min] = temp;
```



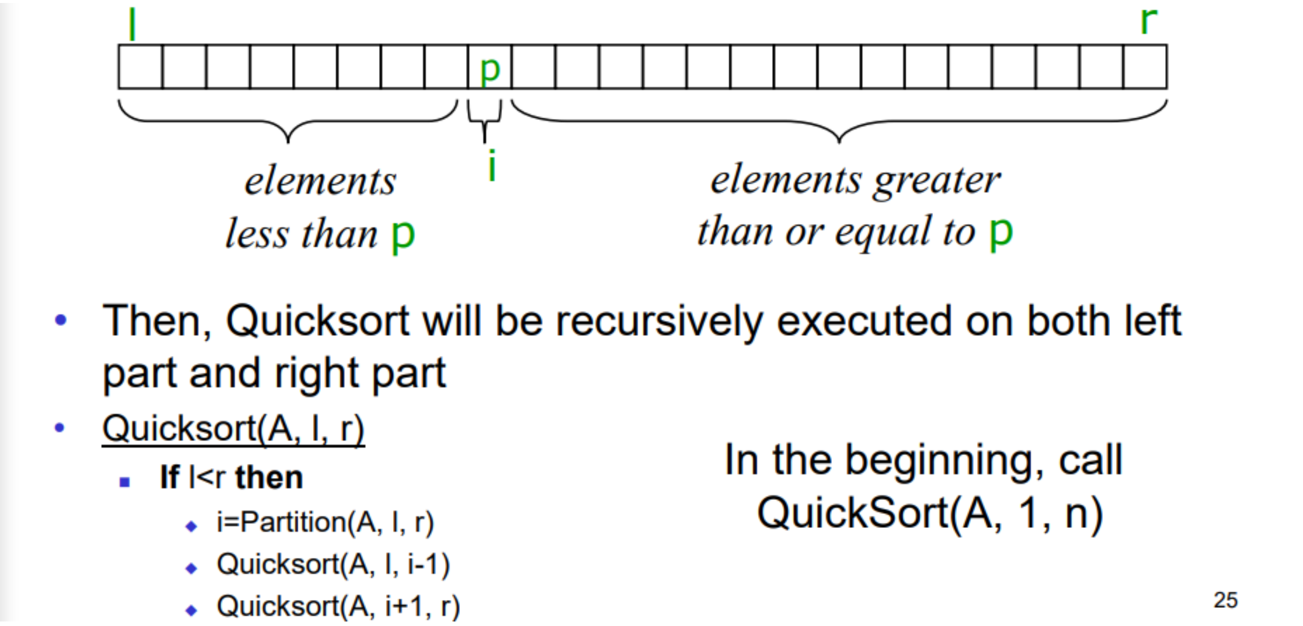
Quick Sort

- Uses divide-and-conquer
- **In-place**
- Very practical, average performance Θ (n log n), but worst case still Θ (n^2).
- Divide:
  - Pick an random element, called a pivot, from the array.
  - Reorder the array so that
    - All elements which are less than the pivot come before the pivot (i.e., on the left side of the pivot), and
    - All elements greater than the pivot come after it (i.e., on the right side of the pivot).
    - Equal values can go either way.
    - After this partitioning, the pivot is in its final position.
  - This is called partition operation
- Conquer:
  - Recursively call quick sort to sort the 2 subarrays
- Combine:
  - Trival since sorting is done in place.
- Key Characteristics

- The divide-and-conquer nature is like merge sort, but it does not require an additional array.
    - It sorts **in-place**.
- Very practical, average performance $\Theta(nlogn)$, but worst case still $\Theta(n^2)$.

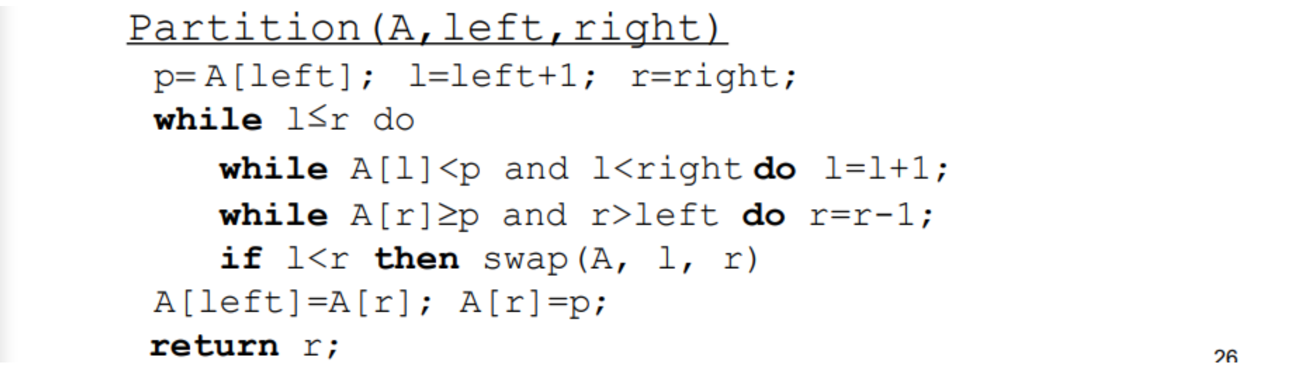
Partitioning: Key Step in Quicksort

- Choose some (any) element **p** in the array as a pivot.
- Partition the array into three parts based on the pivot.
    - Left part, the pivot itself, and right part
    - Partition returns the final index of p in the array

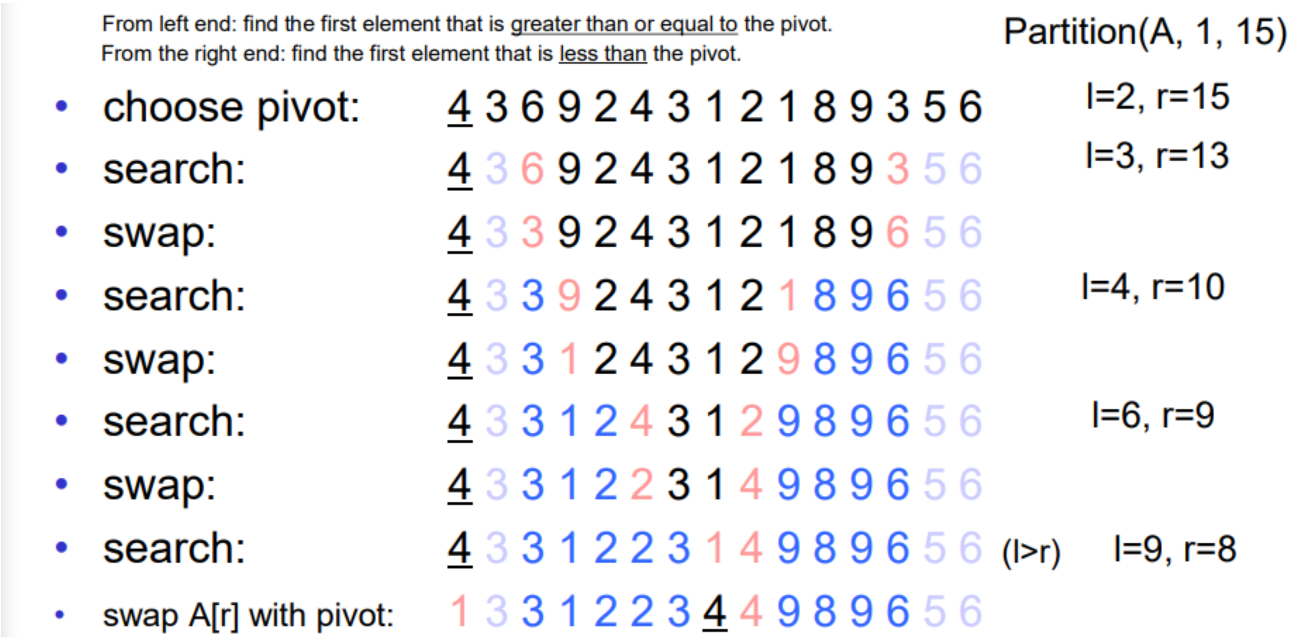


Partition Algorithm

- Choose an array element (say, the first) to use as the pivot.
- Starting from the left end, find the first element that is **greater than or equal to** the pivot.
- Searching backward from the right end, find the first element that is **less than** the pivot.
- Swap these two elements.
- Repeat, searching from where we left off, until all elements are checked.



Example of Partitioning



- The run time of partition is Θ(n).
  - Just need to go through the whole array.

Mini quiz

- Try quick sort on the following array
- 3, 0, 1, 8, 7, 2, 5, 4, 9, 6

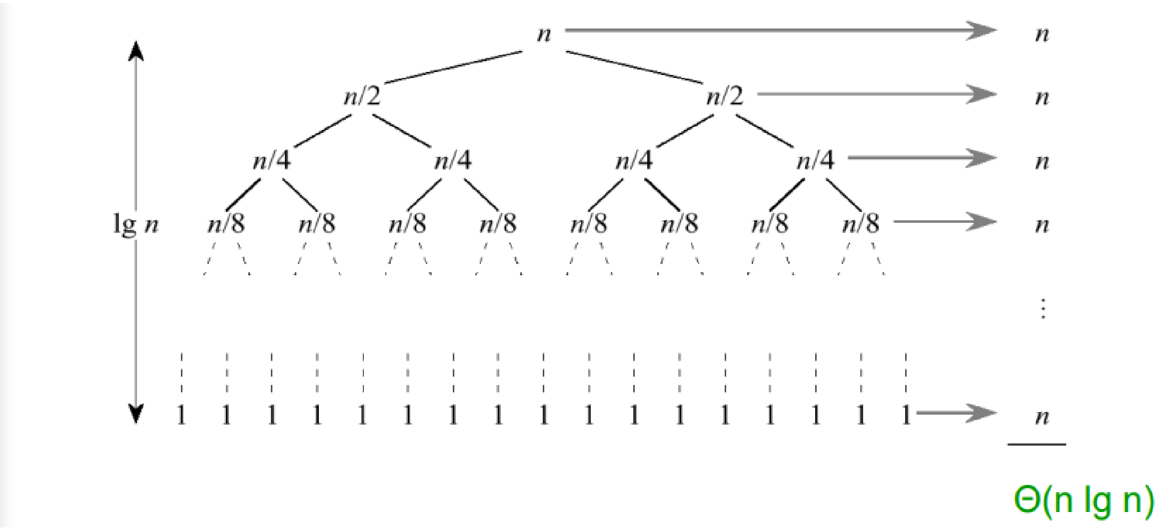
- How does it look after the first call of partition?
- 3, 0, 1, **8**, 7, **2**, 5, 4, 9, 6 (search, l=4, r=6)
- 3, 0, 1, **2**, 7, **8**, 5, 4, 9, 6 (swap)
- 3, 0, 1, **2**, **7**, 8, 5, 4, 9, 6 (search, l=5, r=4)
- 2, 0, 1, **3**, 7, 8, 5, 4, 9, 6 (swap A[4] with pivot)

Analysis of Quicksort

- The running time depends on the distribution of splits.

Base Case Partitioning

- If we are lucky, Partition always splits the array evenly

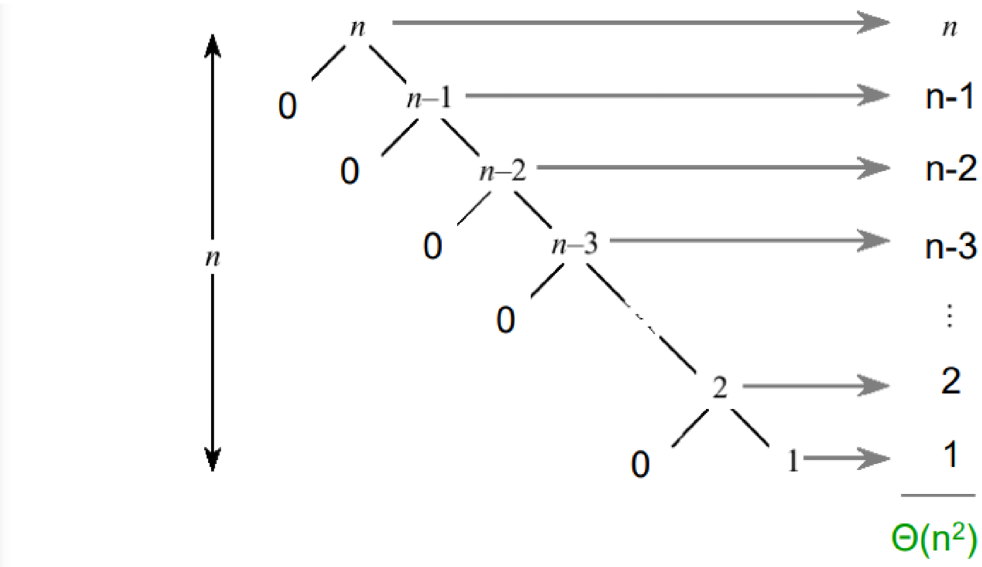


$T(n) - 2T(n/2) + Q(n)$

Worst Case of Quicksort

- In the worst case, partitioning always divides the size **n** array into these three parts:
  - A length zero part, and
  - A length one part, containing the pivot itself
  - A length **n-1** part, containing everything else

Worst Case Partitioning

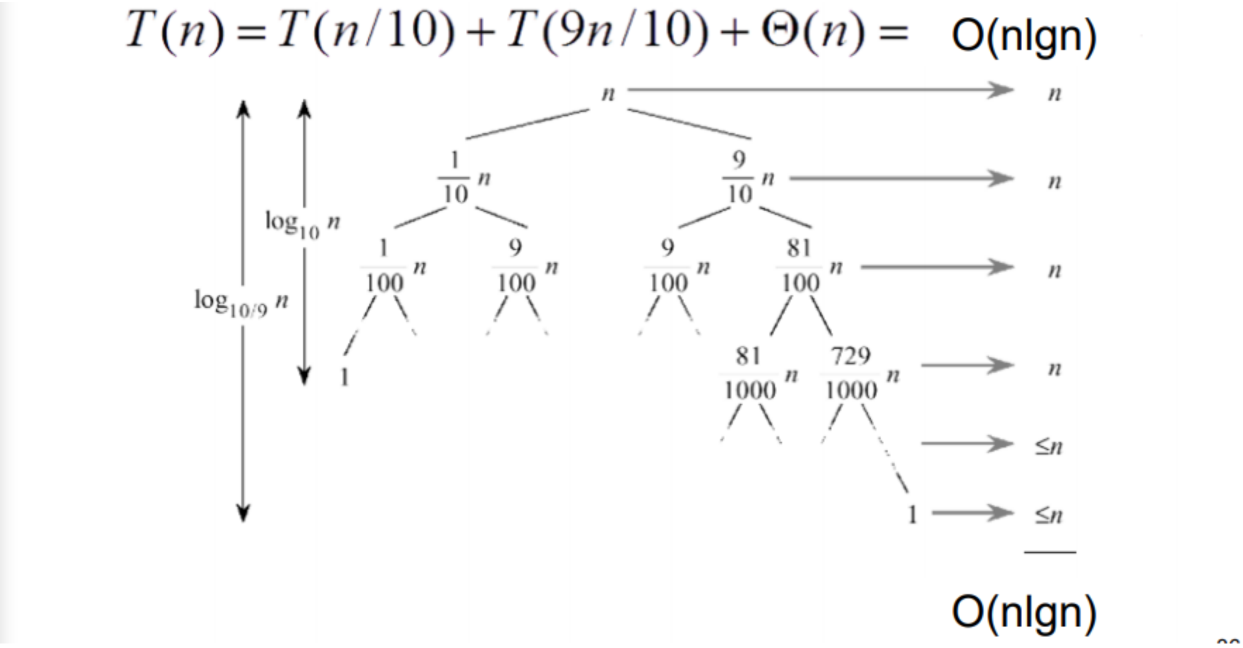


Mini quiz

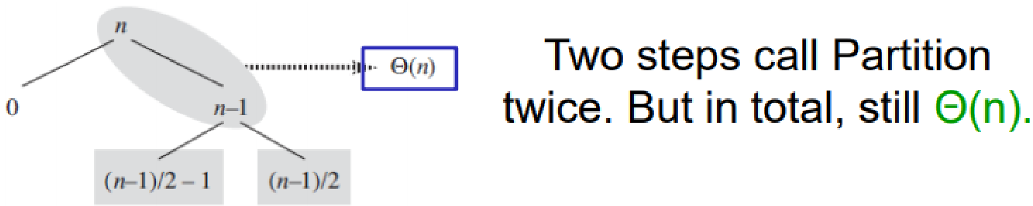
- Can you write down the recurrences for the worst case for quick sort?
  - $T(n) = T(n-1) + \Theta(n)$
- When does the worst case happen?
  - Input array is sorted.
  - Input array is inversed sorted.
- Note that when the input array is sorted, insertion sort is in the best case that has run time Θ(n).

How about average case?

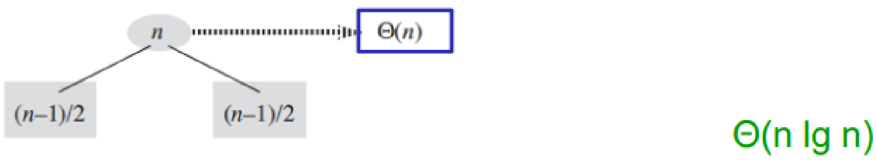
- Average case run time is much closer to the best case than to the worst case.
  - Assume we have balanced partition, e.g., 1-to-9 split.



- Any split of constant proportionality yields a recursion tree of depth Θ(lgn)
  - E.g., 1-to-9, 1-to-99, ...,1-to-999999, ...
- Per level cost is at most n, i.e., O(n)
- If it is not the worst case, always O(nlgn).
- Assume that we are unlucky and then lucky.
- Worst case and then best case.



- Think two steps together, we get the following recursion tree.



Picking a Better Pivot

- So far, we picked the *first* element of each sub-array to use as a pivot
  - If the array is already sorted, this results in O(n2) complexity
  - It's no better if we pick the *last* element
- We could do an optimal quicksort (guaranteed Θ(nlgn)) if we always picked a pivot value that exactly cuts the array in half

- Such a value is called a median
  - half of the values in the array are larger, half are smaller
- The easiest way to find the median is to *sort* the array and pick the value in the middle (!)
  - Ironically
- Random pivot
  - Randomized algorithm of partitioning

```
Randomized-Partition(A,left,right)
i←Random(left,right)
exchange A[left]↔A[i]
return Partition(A,left,right)
Randomized-Quicksort(A,p,r)
if p<r then
    q←Randomized-Partition(A,p,r)
    Randomized-Quicksort(A,p,q-1)
    Randomized-Quicksort(A,q+1,r)
```

	Worst case run time	**Average case run time**	**In place or not?**
Merge sort	$\Theta(nlgn)$	$\Theta(nlgn)$	No. Requires $\Theta(n)$ additional storage.
Insertion sort	$\Theta(n^2)$	$\Theta(n^2)$	Yes. Requires constant additional storage.
Bubble sort	$\Theta(n^2)$	$\Theta(n^2)$	Yes.
Selection sort	$\Theta(n^2)$	$\Theta(n^2)$	Yes.
Quick sort	$\Theta(n^2)$	$\Theta(nlgn)$	Yes.

Exam 2015

3. (*5 points*) Let's consider a scenario that Jakob uses a sorting algorithm to sort the following numbers (84, 45, 22, 11, 21). The algorithm produces the following sequences of numbers as it proceeds:
(11, 45, 22, 84, 21), (11, 21, 22, 84, 45), (11, 21, 22, 84, 45), (11, 21, 22, 45, 84).
The sorting algorithm that Jakob used is:

☐ **a)** Quick Sort ☒ **b)** Selection Sort
☐ **c)** Insertion Sort ☐ **d)** Bubble Sort