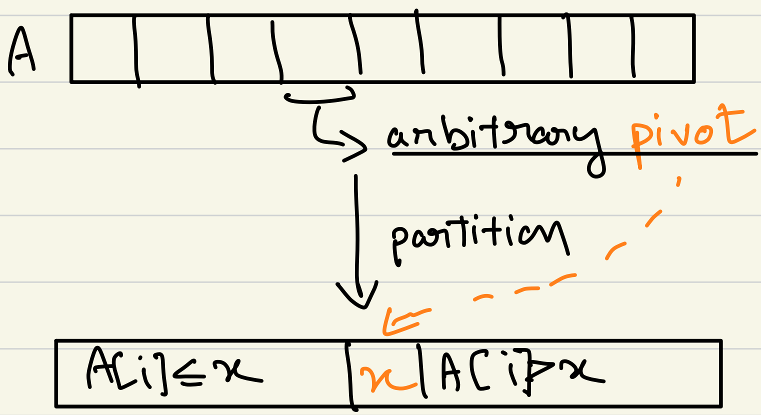
Quick Sort & Heap Sort

▷ Best of both worlds! | O(n log n) runtime & in-place

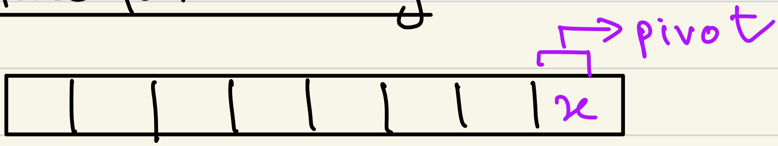
▷ Quicksort

- ▷ Also divide & conquer.
- ▷ Unlike merge-sort, most work done during divide stage.



▷ Partitioning is trivial in time (O(n)) if we don't care about in-place.

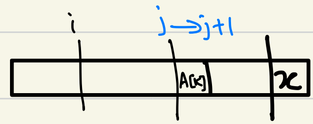
▷ In-place partitioning

A 

i ; $k \le i \rightarrow A[k] \le x$

j ; $k \ge j \rightarrow A[k] > x$

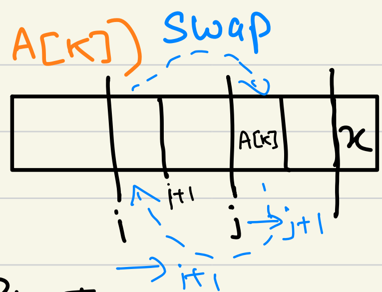
▷ Case 1: $A[k] > x \Rightarrow$ j++



▷ Case 2: $A[k] \le x$

$\Rightarrow$ ① Swap (A[i+1], A[k])

② i++ , j++



▷ Final step: Pivot is ≤ Pivot

$\Rightarrow$ ① Swap (A[pivot], A[j])

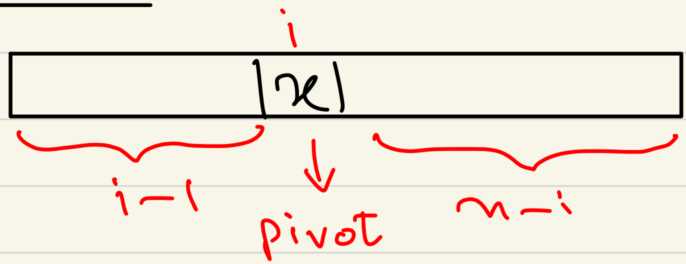
② j++

(D) Pseudocode:

```
partition (A, p, r) {
    pivot ← A[r]
      i ← p-1
      for j=p to r
        if A[j] ≤ x {
            i=i+1
            swap (A[i], A[j])
        }
      return i }

quick-sort (A, p, r) {
    if p < r {
        i ← partition (A, p, r)
        quick-sort (A, p, i-1)
        quick-sort (A, i+1, r)
    }
}
```

⊳ Runtime



$$T(n) = T(i-1) + T(n-i) + n$$

⊳ Case 1: $i = 1$ (Worst Case)

$$T(n) = T(n-1) + n = \Theta(n^2)$$

⊳ Case 2: $i = n/2$

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

$$\boxed{\text{⊳ The runtime is too imbalanced due to poor worst case.}}$$

How do we mitigate this?

▷ RANDOMIZED QUICKSORT

▷ Use randomized pivot to gain immunity from worst case (reverse sorted array)

▷ Pseudocode

```
RQS(A, p, r) {

    pick random j from [p, r]
    Swap (A[j], A[r])            → Random pivot

    if p < r {
        i ← partition (A, p, r)

        RQS (A, p, i-1)
        RQS (A, i+1, r)
    }
}
```

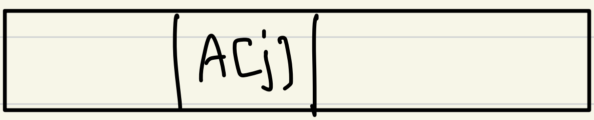
▷ Runtime

Since we randomize the pivot, worst case is no more or less worse than average (random) case

$\Rightarrow T(n) = T(n)_{\text{average}}$

$\Rightarrow T(n) = \text{avg.}\left(T(i-1) + T(n-i) + n\right)$

pivot $\Rightarrow$ random var.



$P(j = i \in [1 \ldots n]) = \frac{1}{n}$

$\boxed{E[x] = P(X = x) \cdot x}$

$\Rightarrow T(n) = \frac{1}{n}\left(T(0) + T(n-1) + n\right)$
$+ \frac{1}{n}\left(T(1) + T(n-2) + n\right)$
$\vdots$
$+ \frac{1}{n}\left(T(n-1) + T(0) + n\right)$

$$= \frac{1}{n}\Big[\big(T(0)+T(n-1)\big)+\big(T(1)+T(n-2)\big)$$

$$+\dots+\big(T(n-2)+T(1)\big)+$$

$$\big(T(n-1)+T(0)\big)\Big]+n$$

pairs

$$= \frac{2}{n}\big(T(0)+T(1)+\dots T(n-1)\big)+n$$

$$\Rightarrow n\cdot T(n) = 2\sum_{k=0}^{n-1} T(k) + n^2 \rightarrow (1)$$

$$\Rightarrow (n-1)T(n-1) = 2\sum_{k=0}^{n-2} T(k) + (n-1)^2 \rightarrow (2)$$

$$(1) - (2)$$

$$\Rightarrow n\cdot T(n) - (n-1)T(n-1) =$$

$$2T(n-1) + \big(n^2 - (n-1)^2\big)$$

$$\Rightarrow nT(n) = (n+1)T(n-1) + 2n-1$$

$$\Rightarrow \frac{T(n)}{n+1} = \frac{T(n-1)}{n} + \frac{2n-1}{n(n+1)} \leq \frac{T(n-1)}{n} + \frac{2}{n}$$

Let $S(n) = \frac{T(n)}{n+1}$

$$\Rightarrow S(n) = S(n-1) + \frac{2}{n} = \frac{2}{1} + \frac{2}{2} + \dots \frac{2}{n}$$

$$= 2\underbrace{\left(1 + \frac{1}{2} + \frac{1}{3} + \dots + \frac{1}{n}\right)}_{\text{Harmonic Series}}$$

$$= \Theta(\log n)$$

$$T(n) = (n+1)\, S(n)$$

$$\boxed{T(n) = \Theta(n \log n)}$$

▷ HEAPSORT

▷ Recall Selection sort:

$i \leftarrow n$

while $i > 1$

▷ Find $\max(A[1..i])$

▷ Call it $A[j]$

▷ Swap $(A[i], A[j])$

▷ $i \leftarrow i-1$

end while

$\boxed{\text{Optimize extraction of max element}}$

What data structures can we use to "extract max element"?

▷ Priority Queue (PQ)

1. $S \leftarrow \text{BuildPQ}(A, n)$
2. $\text{Insert}(S, x)$

(3) $\mathrm{Max}(S) \longrightarrow z$

(4) $\mathrm{ExtractMax}(S) \longrightarrow z$

(5) $\mathrm{IncreaseKey}(S, x, K)$

if $x.key \leq K$, set $x.key = K$

PQSort (A, n)

$S \leftarrow \mathrm{BuildPQ}(A, n)$

for $i = n$ to 1

$A[i] \leftarrow \mathrm{Extractmax}(S)$

endfor

$\mathrm{Time}(\mathrm{PQSort}, n) \leq \mathrm{Time}(\mathrm{BuildPQ}, n)$

$+$

$n \cdot \mathrm{Time}(\mathrm{ExtractMax}, n)$

Case 1: Unsorted array

BuildPQ : 0

ExtractMax: n

$\downarrow$

Selection sort

Case 2: Sorted Array

BuildPQ : $n \log n$ using MS
ExtractMax : $O(1)$ (return $A[n]$)

$$\Rightarrow \quad T_{sorting} = T_{BuildPQ} + T_{ExtractMax}$$
$$= n \log n + n$$
$$= O(n \log n)$$

But we just used another sorting algo.

Case 3: Heap Sort

1. $S \leftarrow$ BuildHeap (A, n) | $O(n)$
2. Insert (S, x) record with x.key
3. Max $(S) \rightarrow z$
   or
   ExtractMax $(S) \rightarrow z$ $O(\log n)$
4. IncreaseKey (S, x, k)
   // if $x.key \leq k$, set $x.key = k$

Time(Heapsort, n) $= n + n \log n = O(n \log n)$

▷ Since heap is in-place PQ, so is Heapsort