COT 5407: Introduction to Algorithms

Giri Narasimhan

ECS 254A; Phone: x3748

giri@cis.fiu.edu

http://www.cis.fiu.edu/~giri/teach/5407S17.html

https://moodle.cis.fiu.edu/v3.1/course/view.php?id=1494

Recap of Sorting Algorithms

- SelectionSort
- InsertionSort
- BubbleSort
- QuickSort
- MergeSort
- HeapSort
- Bucket & Radix Sort
- Counting Sort

Worst Case: $O(N^2)$ (SelectionSort, InsertionSort, BubbleSort)

Avg Case: O(N log N) (QuickSort)

Worst Case: O(N logN) (MergeSort, HeapSort)

Lower Bound for Comparison-based Sorting

Worst Case: O(N); Not comparison-based (Bucket & Radix Sort, Counting Sort)

Tree Sorting

- BST is a search structure that helps efficient search
  - Search can be done in O(h) time, where h = height of BST
  - Also inserts and deletes can be done in O(h) time
  - Unfortunately, Height h = O(N)
- **Balanced** BST improves BST with h = O(log N)
  - Thus search can be done in O(log N)
  - And, inserts and deletes too can be done in O(log N) time
- We can use **B**BSTs in the following way:
  - Repeatedly insert N items into a **B**BST
  - Repeatedly delete the smallest item from the BBST until it is empty
- N inserts and N deletes can be done in O(N log N) time

Order Statistics

- Maximum, Minimum

7	3	1	9	4	8	2	5	0	6

  - Upper Bound
    - O(n) because
    - We have an slgorithm with a single for-loop: n-1 comparisons
  - Lower Bound
    - n-1 comparisons
- MinMax
  - Upper Bound: 2(n-1) comparisons
  - Lower Bound: 3n/2 comparisons
- Max and 2ndMax
  - Upper Bound: (n-1) + (n-2) comparisons
  - Lower Bound: Harder to prove

<u>Rank</u>$_A$(x) = position of x in sorted order of A

k-Selection; Median

- Select the k-th smallest item in list
- Naïve Solution
  - Sort;
  - pick the k-th smallest item in sorted list.

    O(n log n) time complexity
- Idea: Modify Partition from QuickSort
  - How?
- Randomized solution: Average case O(n)
- Improved Solution: worst case O(n)

Using Partition for k-Selection

- Perform Partition from QuickSort (assume all unique items)
- Rank(pivot) = 1 + # of items that are smaller than pivot
- If Rank(pivot) = k, we are done
- Else, recursively perform k-Selection in one of the two partitions

```
PARTITION(array A, int p, int r)
x ← A[r]                              ▷ Choose pivot
i ← p − 1
for j ← p to r − 1
      do if (A[j] ≤ x)
            then i ← i + 1
                  exchange A[i] ↔ A[j]
exchange A[i + 1] ↔ A[r]
return i + 1
```

QuickSelect: a variant of QuickSort

```
QUICKSELECT(array A, int k, int p, int r)
    ▷ Select k-th largest in subarray A[p..r]
    if (p = r)
        then return A[p]
    q ← PARTITION(A, p, r)
    i ← q − p + 1          ▷ Compute rank of pivot
    if (i = k)
        then return A[q]
    if (i > k)
        then return QUICKSELECT(A, k, p, q)
        else return QUICKSELECT(A, k − i, q + 1, r)
```

k-Selection Time Complexity

- Perform Partition from QuickSort (assume all unique items)
- Rank(pivot) = 1 + # of items that are smaller than pivot
- If Rank(pivot) = k, we are done
- Else, recursively perform k-Selection in one of the two partitions

- On the average:
  - Rank(pivot) = n / 2
- Average-case time
  - T(N) = T(N/2) + O(N)
  - T(N) = O(N)
- Worst-case time
  - T(N) = T(N-1) + O(N)
  - $T(N) = O(N^2)$

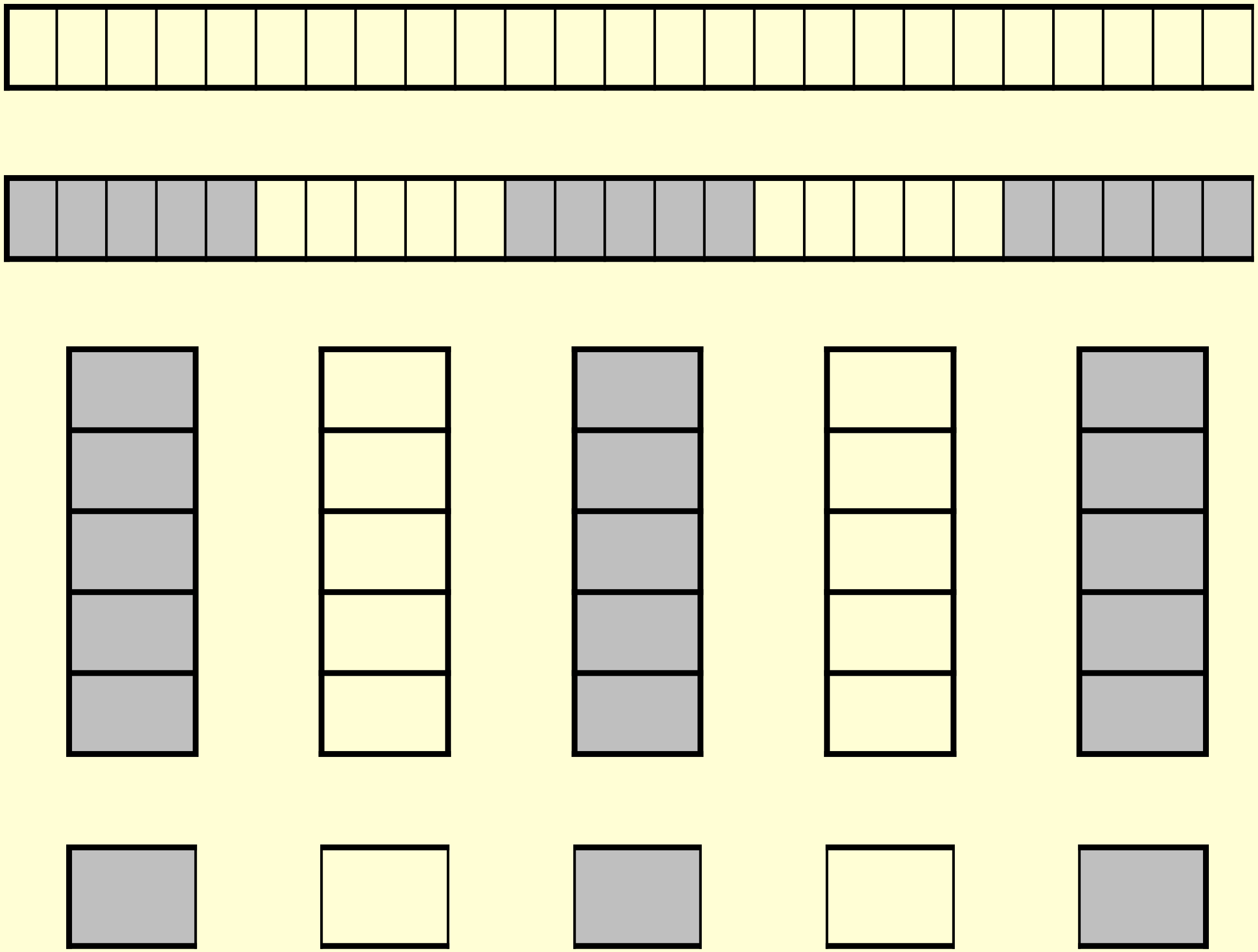
```
PARTITION(array A, int p, int r)
x ← A[r]                              ▷ Choose pivot
i ← p − 1
for j ← p to r − 1
        do if (A[j] ≤ x)
                then i ← i + 1
                     exchange A[i] ↔ A[j]
exchange A[i + 1] ↔ A[r]
return i + 1
```

Randomized Solution for k-Selection

- Uses RandomizedPartition instead of Partition
  - RandomizedPartition picks the pivot uniformly at random from among the elements in the list to be partitioned.
- Randomized k-Selection runs in $O(N)$ time on the average
- Worst-case behavior is very poor $O(N^2)$

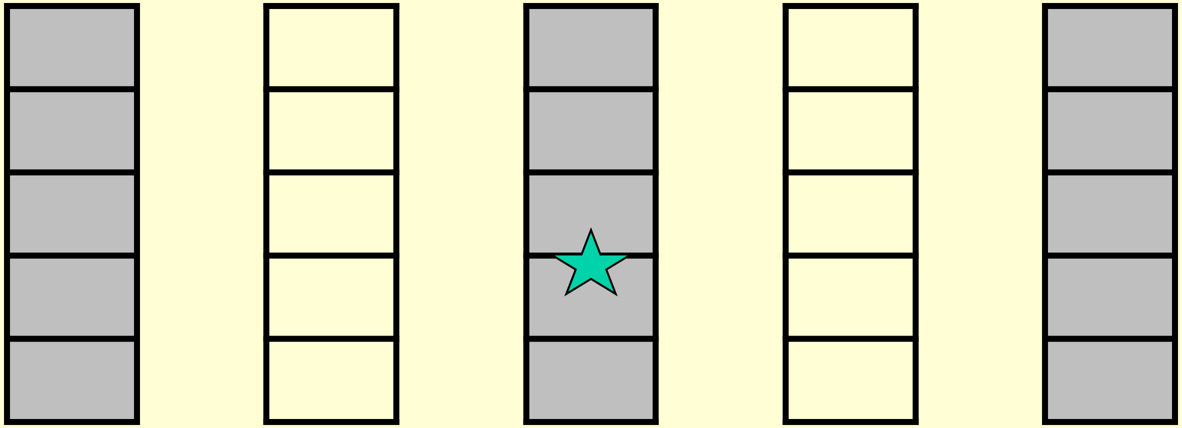
k-Selection & Median: Improved Algorithm

- Start with initial array



k-Selection & Median: Improved Algorithm(Cont'd)

- Use median of medians as pivot



- $T(n) < O(n) + T(n/5) + T(3n/4)$

ImprovedSelect

```
ImprovedSelect(array A, int k, int p, int r)
    ▷ Select k-th largest in subarray A[p..r]
    if (p = r)
        then return A[p]
        else N ← r − p + 1
    Partition A[p..r] into subsets of 5 elements and
    collect all medians of subsets in B[1..⌈N/5⌉].
    Pivot ← ImprovedSelect(B, 1, ⌈N/5⌉, ⌈N/10⌉)
    q ← PivotPartition(A, p, r, Pivot)
    i ← q − p + 1        ▷ Compute rank of pivot
    if (i = k)
        then return A[q]
    if (i > k)
        then return ImprovedSelect(A, k, p, q − 1)
        else return ImprovedSelect(A, k − i, q + 1, r)
```

PivotPartition

PivotPartition(*array* A, *int* p, *int* r, *item Pivot*)

$\triangleright$ Partition using provided *Pivot*

$i \leftarrow p - 1$

for $j \leftarrow p$ **to** r

  do if $(A[j] \leq Pivot)$

    then $i \leftarrow i + 1$

      exchange $A[i] \leftrightarrow A[j]$

return $i + 1$