

DSC40B:
Theoretical Foundations of Data
Science II

Lecture 7: *The Median, order statistics,
QuickSelect and QuickSort*

Instructor: Yusu Wang

Previously

- ▶ Sorting an array
- ▶ (Binary) search in a sorted array

- ▶ Today:
 - ▶ What if, without sorting, we would like to select a specific number with a certain rank in the array
 - ▶ For example, how to find the median of **an unsorted** array of numbers quickly?



Part A:
Order statistics and simple examples



Order statistics

- ▶ Given a set of n numbers
 - ▶ The **k th order statistics** is the k th smallest number in this collection
 - ▶ We also say that this number has **rank k** in the input.
- ▶ Examples:
 - ▶ 1st order statistics: minimum
 - ▶ n th order statistics: maximum
 - ▶ $\lceil \frac{n}{2} \rceil$ -th order statistics: median
 - ▶ $\lceil \frac{pn}{100} \rceil$ -th order statistics: p -th percentile



Select problem

- ▶ Input: given n numbers stored in an array A , and an order (rank) $k \in [1, n]$
- ▶ Output: return the k -th order statistics of A

- ▶ Special cases:
 - ▶ $k = 1$? $k = n$?
 - ▶ But how about for general k , including finding the median of A ?



Simple approaches

- ▶ Approach 1:
 - ▶ Modifying selection sort
 - ▶ Stops when find the k -th order statistics



Algorithm selection_sort

```
def selection_sort(A):  
    n = len(A)  
    if n <= 1:  
        return  
    for barrier_id in range(n-1):  
        # find index of min in A[start:]  
        min_id = find_minimum(A, start=barrier_id)  
        #swap  
        A[barrier_id], A[min_id] = (  
            A[min_id], A[barrier_id]  
        )
```



Algorithm selection_kthOS

```
def selection_kthOS(A, k):  
    n = len(A)  
    if n < k:  
        return Error  
    for barrier_id in range(k):  
        # find index of min in A[start:]  
        min_id = find_minimum(A, start=barrier_id)  
        #swap  
        A[barrier_id], A[min_id] = (  
            A[min_id], A[barrier_id]  
        )  
    return A[k-1]
```



Simple approaches

- ▶ Approach 1:

- ▶ Modifying selection sort
- ▶ Stops when find the k -th order statistics
- ▶ Time complexity
 - ▶ $\Theta(kn)$

- ▶ Approach 2:

- ▶ First sort array A
- ▶ Return $A[k]$
- ▶ Time complexity
 - ▶ Same as sorting, which is $\Theta(n \lg n)$

Can we do better than sorting
(namely $\Theta(n \lg n)$ time)?



Part B:
Can we do better than sorting?
First try of *QuickSelect*

I will use pseudo-code in what follows.
As convention: array index starts from 0.



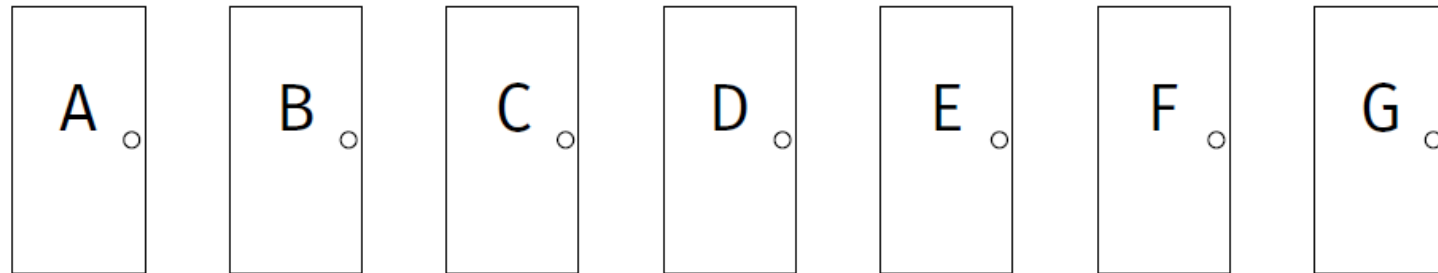
Select problem

- ▶ Input: given n numbers stored in an array A , and an order $k \in [1, n]$
- ▶ Output: return the k -th order statistics of A
- ▶ Intuition:
 - ▶ In Sorting, we essentially figure out the relative orders among all elements
 - ▶ There is much redundancy; for example, if two numbers both have higher order than the target order k , then intuitively, we don't care about spending time to figure out their relative order.
 - ▶ So intuitively, we should be able to do better than sorting.
 - ▶ How to leverage this thought?



An example

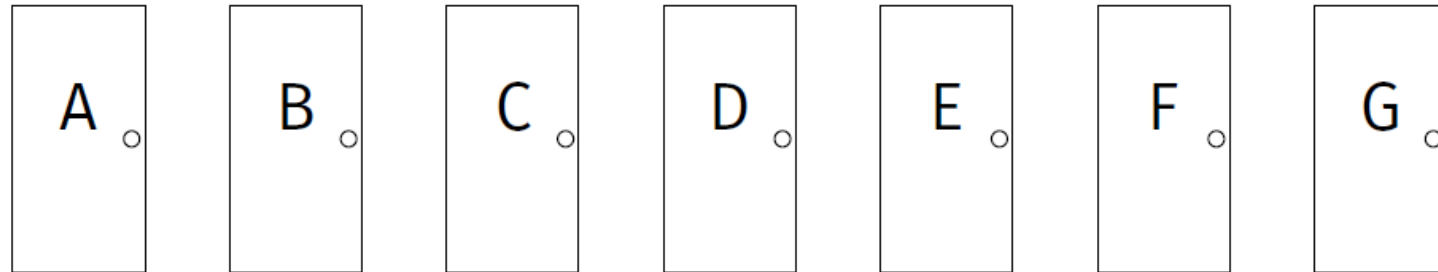
- ▶ Given n doors, need to find the **largest number** behind the door
- ▶ Each time we open a door, we have an oracle to tell us
 - ▶ which doors are smaller, and
 - ▶ which doors are bigger

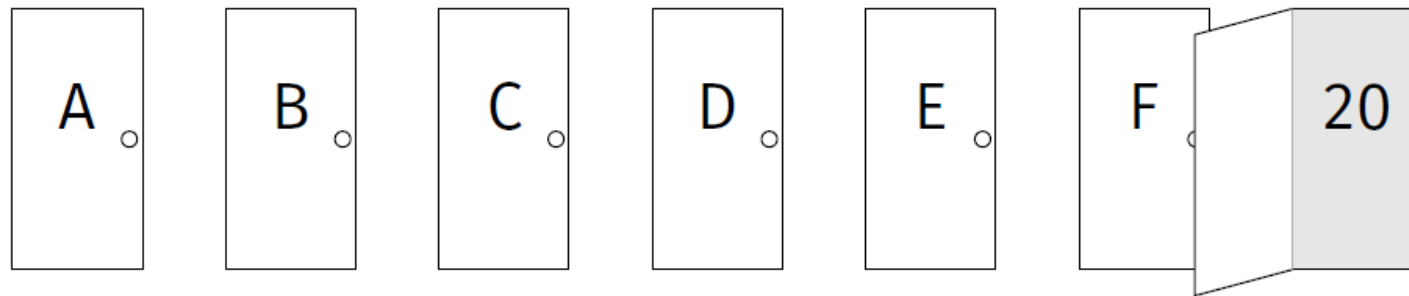


An example

- ▶ Given n doors, need to find the **largest number** behind the door
- ▶ Each time we open a door, we have an oracle to tell us
 - ▶ which doors are smaller, and
 - ▶ which doors are bigger

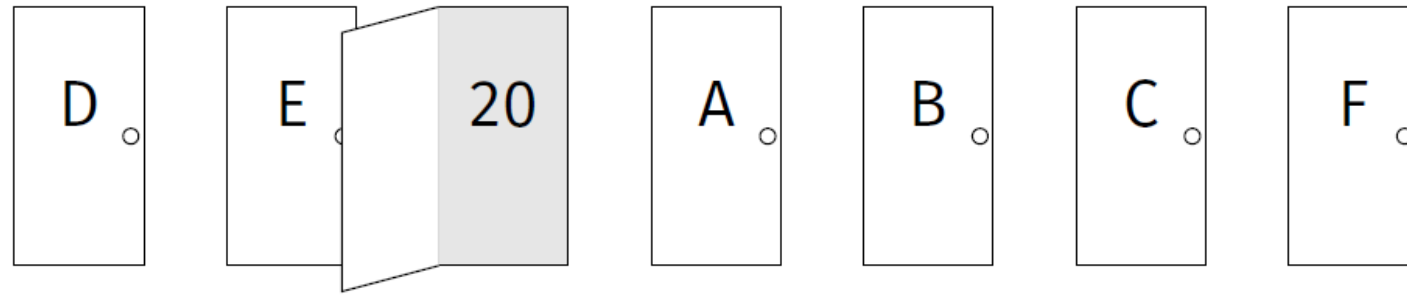
Call this a **partition** operation





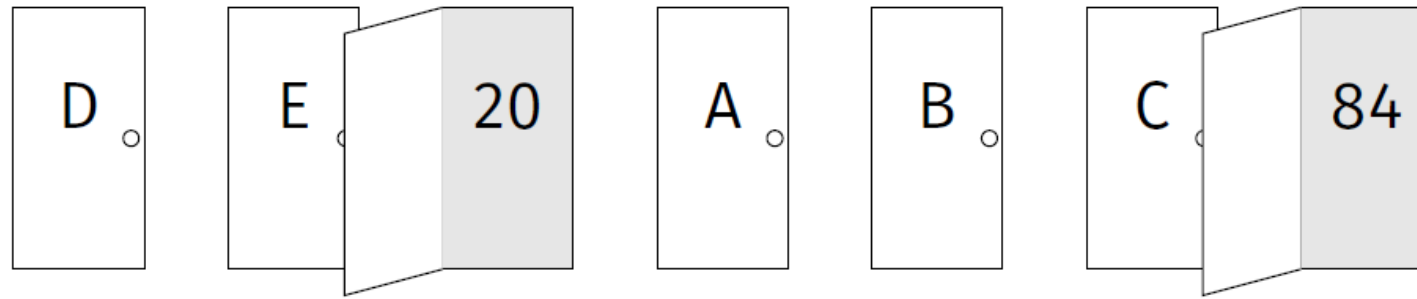
we open the last door





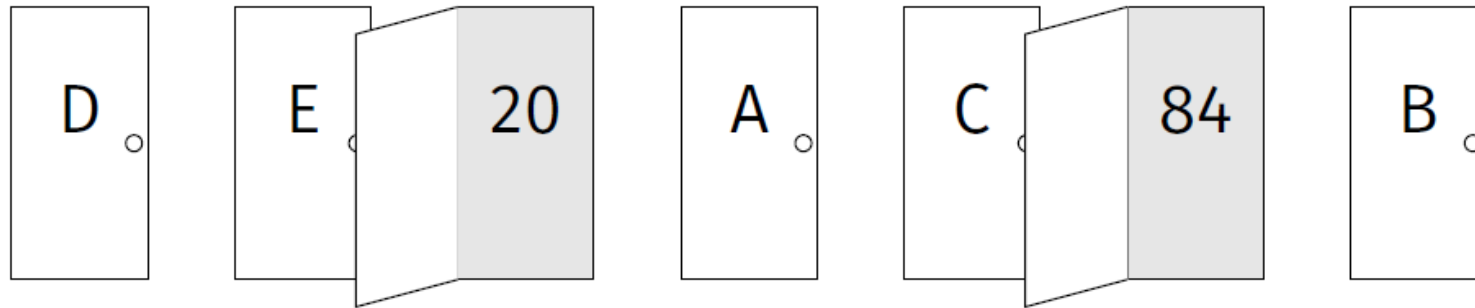
after partition





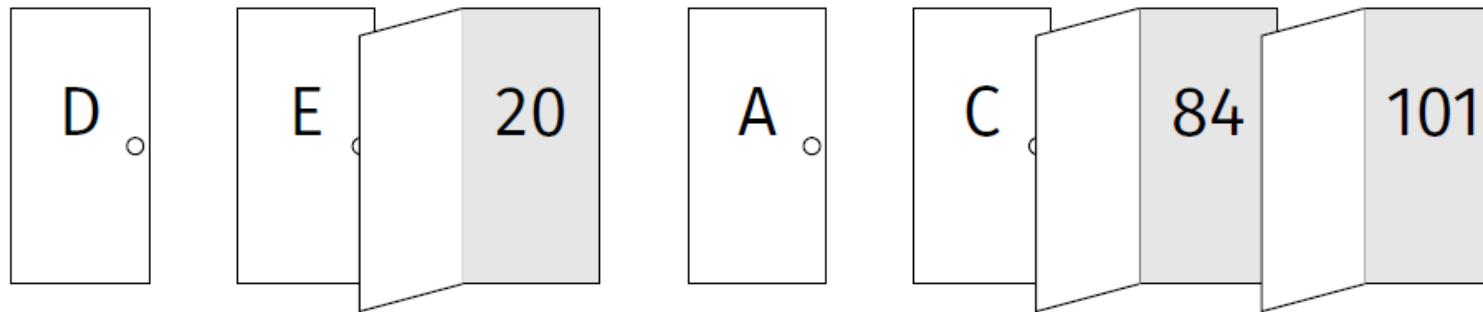
repeat in the right portion:
open the last door of this subarray





repeat in the right portion:
after **partition** in this subarray





again, go to the right portion:
only 1 entry left: must be the largest,
and we return



Generalizing the idea?

- ▶ Assume that we are given the Partition procedure:

- ▶ **Partition** (A, s, t)

- ▶ Input:

- ▶ Given an array A and consider sub-array $A[s, \dots t - 1]$
 - ▶ $A[t - 1]$ will be used as **the pivot** $p = A[t - 1]$

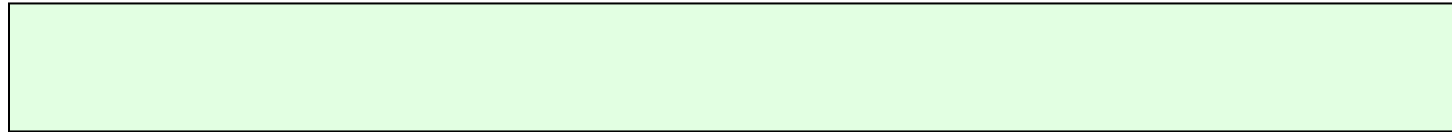
- ▶ Output:

- ▶ Rearrange elements in A where p is now in $A[m]$ such that
 - all elements $\leq p$ are to its left
 - all elements $> p$ are to its right
 - ▶ Return the new position **m** of the pivot p



Intuition of QuickSelect

- ▶ Suppose we are already given the **Partition** procedure.
- ▶ $\text{QuickSelect}(A, 0, n, k)$
 - ▶ $m = \text{Partition}(A, 0, n)$
 - ▶ Note: the **order** of the pivot = $m + 1$



Intuition of QuickSelect

- ▶ Suppose we are already given the **Partition** procedure.
- ▶ $\text{QuickSelect}(A, 0, n, k)$
 - ▶ $m = \text{Partition}(A, 0, n)$
 - ▶ Note: the **order** of the pivot = $m + 1$



$p = A[m]$: pivot



Intuition of QuickSelect

- ▶ Suppose we are already given the **Partition** procedure.
- ▶ $\text{QuickSelect}(A, 0, n, k)$
 - ▶ $m = \text{Partition}(A, 0, n)$
 - ▶ Note: the **order** of the pivot = $m + 1$



$p = A[m]$: pivot

Case 1: $k = m+1$

return $A[m]$



Intuition of QuickSelect

- ▶ Suppose we are already given the **Partition** procedure.
- ▶ $\text{QuickSelect}(A, 0, n, k)$
 - ▶ $m = \text{Partition}(A, 0, n)$
 - ▶ Note: the **order** of the pivot = $m + 1$



$p = A[m]$: pivot

Case 2: $k < m+1$



Intuition of QuickSelect

- ▶ Suppose we are already given the **Partition** procedure.
- ▶ $\text{QuickSelect}(A, 0, n, k)$
 - ▶ $m = \text{Partition}(A, 0, n)$
 - ▶ Note: the **order** of the pivot = $m + 1$



$p = A[m]$: pivot

Case 2: $k < m+1$

return $\text{QuickSelect}(A, 0, m, k)$



Intuition of QuickSelect

- ▶ Imagine we are given **Partition** procedure.
- ▶ $\text{QuickSelect}(A, 0, n, k)$
 - ▶ $m = \text{Partition}(A, 0, n)$
 - ▶ Note: the **order** of the pivot = $m + 1$



$p = A[m]$: pivot

Case 3: $k > m+1$

return $\text{QuickSelect}(A, m+1, n, k)$



Pseudo-code for QuickSelect

```
QuickSelect ( A, s, t, k )  
    /* select the order k element in A from subarray A[s,..t-1] */  
    if ( k < s or k ≥ t or s ≥ t ) return None;  
    m = Partition ( A, s, t );  
    pivot_order = m+1 ;  
    if ( pivot_order = k ) return A[m];  
    if ( pivot_order > k )  
        return QuickSelect ( A, s, m, k );  
    else return QuickSelect ( A, m+1, t, k );
```

At the top level, we call QuickSelect(A, 0, n, k)



Example

- ▶ $A = [13, 2, 5, 9, 4, 6]$
- ▶ Goal: find 2nd order statistics in A ; i.e, $k = 2$



Part C: Partition procedure



Partition procedure

▶ Partition (A, s, t)

▶ Input:

- ▶ Given an array A and consider sub-array $A[s, \dots t - 1]$
- ▶ $A[t - 1]$ will be used as **the pivot** $p = A[t - 1]$

▶ Output:

- ▶ Rearrange elements in A where p is now in $A[m]$ such that
 - all elements $\leq p$ are to its left
 - all elements $> p$ are to its right
- ▶ Return the new position **m** of the pivot p



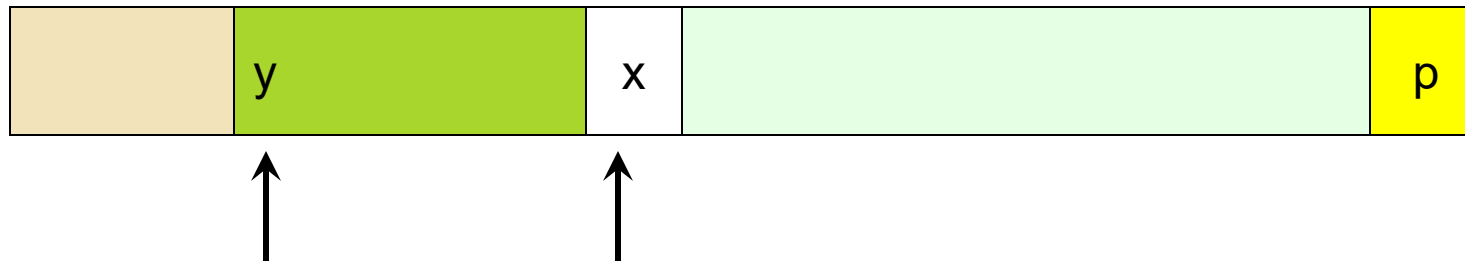
Partition(A, s, t)

Plan: take $A[t-1]$ as pivot

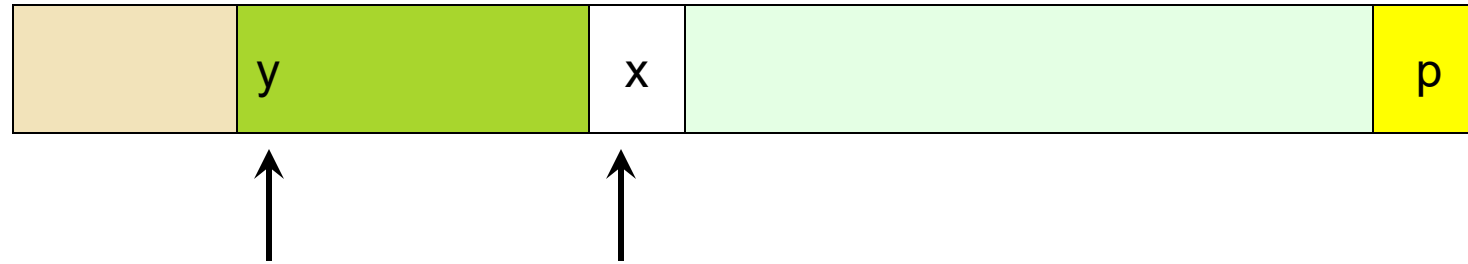


return m

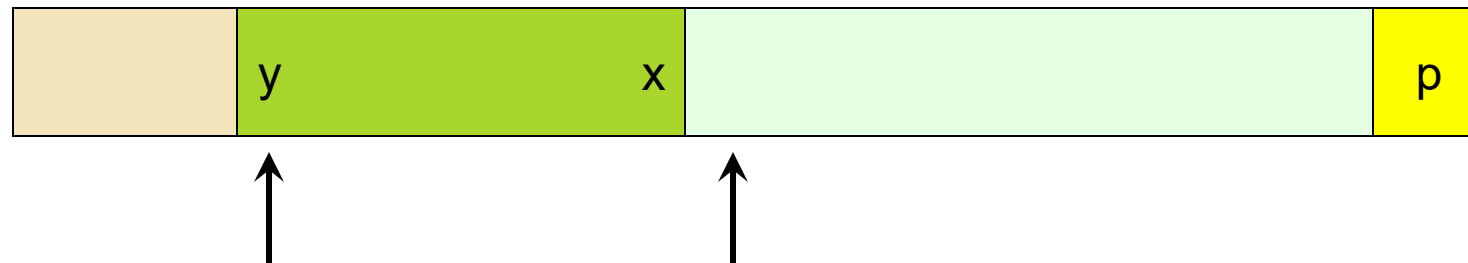
In-place partition !
i.e, we use the same input array,
and only need constant number of auxiliary memory



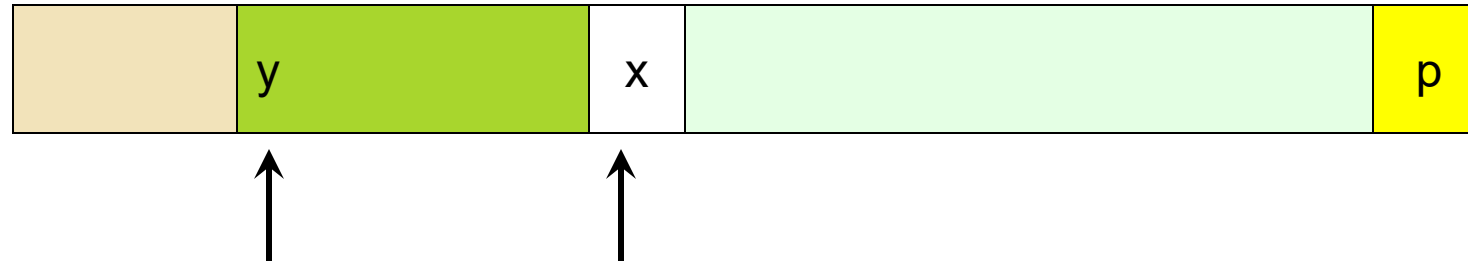
Partition(A, s, t)



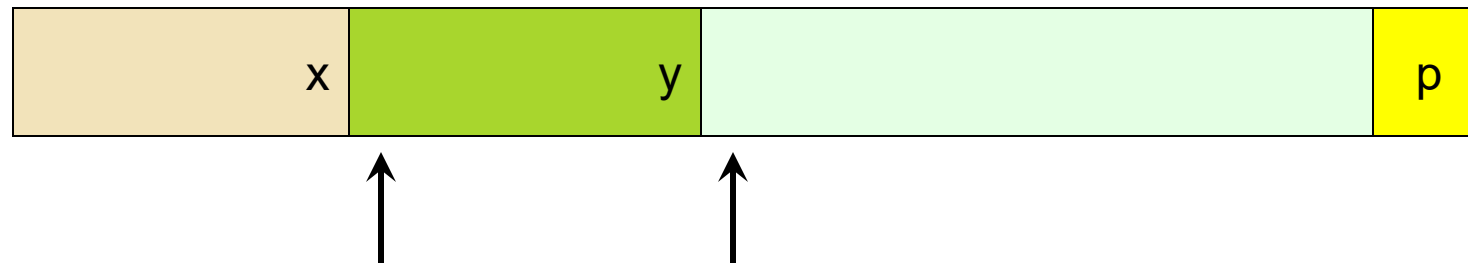
Case 1: $x > p$



Partition(A, s, t)



Case 2: otherwise



► Maintain two pointers:

- “middle” barrier (variable ℓ in code):
 - separates numbers $\leq p$ from those $> p$
 - points to the first number $> p$ so far
- “end” barrier (variable r in code):
 - separates what’s already processed from un-processed
 - points to the first unprocessed number



► **Maintain two pointers:**

► “middle” barrier (variable ℓ in code):

- separates numbers $\leq p$ from those $> p$
- points to the first number $> p$ so far

► “end” barrier (variable r in code):

- separates what’s already processed from un-processed
- points to the first unprocessed number

► **Example:**

[12, 5, 3, 9, 7, 8]



-
- ▶ Example: $A = [12, 5, 3, 9, 7, 8]$

[12, 5, 3, 9, 7, 8]

- ▶ Maintain two pointers:

- ▶ “middle” barrier (variable ℓ in code):

- ▶ separates numbers $\leq p$ from those $> p$
- ▶ points to the first number $> p$ so far

- ▶ “end” barrier (variable r in code):

- ▶ separates what’s already processed from un-processed
- ▶ points to the first unprocessed number



Pseudo-code for Partition

```
Partition( $A, s, t$ )  
/* Partition the subarray  $A[s, \dots, t - 1]$  using  $A[t - 1]$  as pivot.  
/*  $\ell$ : index for mid_barrier; and  $r$ : index for end_barrier.  
  
1  $\ell = s$ ;  
2 for  $r = s$  to  $t - 2$  do  
3   if  $A[r] \leq p$  then  
4     exchange  $A[\ell]$  with  $A[r]$ ;  
5      $\ell++$ ;  
6   end  
7 end  
8 exchange  $A[\ell]$  with  $A[t - 1]$ ;  
9 return ( $\ell$ );
```

In-place!

Time complexity:

$\Theta(t - s)$



Pseudo-code for Partition

```
Partition( $A, s, t$ )  
  /* Partition the subarray  $A[s, \dots, t - 1]$  using  $A[t - 1]$  as pivot.  
  /*  $\ell$ : index for mid_barrier; and  $r$ : index for end_barrier.  
  
  1  $\ell = s$ ;  
  2 for  $r = s$  to  $t - 2$  do  
  3   | if  $A[r] \leq p$  then  
  4   |   | exchange  $A[\ell]$  with  $A[r]$ ;  
  5   |   |  $\ell++$ ;  
  6   | end  
  7 end  
  8 exchange  $A[\ell]$  with  $A[t - 1]$ ;  
  9 return ( $\ell$ );
```



Part D:
Time complexity for QuickSelect
and Randomized QuickSelect



Worst case complexity

```
QuickSelect ( A, s, t, k )  
/* select the order k element in A from subarray A[s,..t-1] */  
if ( k < s or k ≥ t or s ≥ t ) return None;  
m  = Partition ( A, s, t );  
pivot_order = m+1 ;  
if ( pivot_order = k ) return A[m];  
if ( pivot_order > k )  
    return QuickSelect ( A, s, m, k );  
else return QuickSelect ( A, m+1, t, k );
```



Worst case complexity

```
QuickSelect ( A, s, t, k )  
/* select the order k element in A from subarray A[s,..t-1] */  
  if ( k < s or k ≥ t or s ≥ t ) return None;  
  m  = Partition ( A, s, t );  
  pivot_order = m+1 ;  
  if ( pivot_order = k ) return A[m];  
  if ( pivot_order > k )  
    return QuickSelect ( A, s, m, k );  
  else return QuickSelect ( A, m+1, t, k );
```

At the top level, we call QuickSelect(A, 0, n, k).

$T(n) = \max(T(r-1), T(n-r)) + cn$, where $r = m + 1$ is the pivot_order



▶ $T(n) = \max(T(r - 1), T(n - r)) + cn$

▶ Depending on value of r , recursively.

▶ **A lucky case:**

▶ Each time we remove half of the numbers

▶ we cannot do better, why?



▶ $T(n) = \max(T(r - 1), T(n - r)) + cn$

▶ Depending on value of r , recursively.

▶ **A lucky case:**

▶ Each time we remove half of the numbers

▶ we cannot do better, why?

▶ $T(n) = T\left(\frac{n}{2}\right) + cn$
 $= \Theta(n)$



▶ $T(n) = \max(T(r - 1), T(n - r)) + cn$

▶ Depending on value of m , recursively.

▶ **Worst case:**

▶ Each time we can only remove one number

▶ say, the target order $k = n$, while $r - 1$ each time

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



-
- ▶ How to ensure we mostly have “good cases”?
 - ▶ **Good** split:
 - ▶ The pivot splits the current subarray in a balanced way (a constant fraction is on each side, say, the pivot_order r is such that $r \in [\frac{n}{4}, \frac{3n}{4}]$)
 - ▶ **Bad** split:
 - ▶ Otherwise
 - ▶ Roughly speaking, if we always have good splits, then we have that
 - ▶ $T(n) = \Theta(n)$
 - ▶ In fact, this can be relaxed to that if we can have one good split every few (constant number of) splits on average

How to ensure that this happens?



-
- ▶ In other words, when we choose pivot, we hope to choose one whose rank (order) is around the middle
 - ▶ say, between $\frac{n}{4}$ to $\frac{3n}{4}$



-
- ▶ In other words, when we choose pivot, we hope to choose one whose rank (order) is around the middle
 - ▶ say, between $\frac{n}{4}$ to $\frac{3n}{4}$
 - ▶ To guarantee that,
 - ▶ Pick a **random number** in A as the pivot!



-
- ▶ In other words, when we choose pivot, we hope to choose one whose rank (order) is around the middle
 - ▶ say, between $\frac{n}{4}$ to $\frac{3n}{4}$
 - ▶ To guarantee that,
 - ▶ Pick a **random number** in A as the pivot!
 - ▶ Why?
 - ▶ If we pick a random number $x \in A$
 - ▶ i.e, means that the probability of choose any one of the n numbers in A is $\frac{1}{n}$
 - ▶ Probability $\Pr[\text{rank}(x) \in [\frac{n}{4}, \frac{3n}{4}]] = (\frac{3n}{4} - \frac{n}{4}) / n = 2/4 = 1/2$
 - ▶ Hence in expectation, every two times we will have a good split.
-



Rand-Select

```
Rand-Select ( A, s, t, k )
```

```
/* select the order k element in A from subarray A[s,..t-1] */
```

```
if ( k < s or k ≥ t or s ≥ t ) return None;
```

```
m = Rand-Partition ( A, s, t );
```

```
pivot_order = m+1 ;
```

```
if ( pivot_order = k ) return A[m];
```

```
if ( pivot_order > k )
```

```
    return Rand-Select ( A, s, m, k );
```

```
else return Rand-Select ( A, m+1, t, k );
```

Rand-Partition(A, s, t) uses a **random element** from A[s, ... t-1] as pivot, instead of using A[t-1] as pivot like in Partition(A, s, t).



Rand-Partition pseudo-code

Rand-Partition(A, s, t)

/ Partition the subarray $A[s, \dots, t - 1]$ using a random pivot.*

/ ℓ : index for mid_barrier index; and r : index for end_barrier.*

```
1 pivot_id = random(s, t);
2  $p = A[\text{pivot\_id}]$ ;
3 exchange  $A[\text{pivot\_id}]$  with  $A[t - 1]$ ;
4  $\ell = s$ ;
5 for  $r = s$  to  $t - 2$  do
6     if  $A[r] \leq p$  then
7         exchange  $A[\ell]$  with  $A[r]$ ;
8          $\ell++$ ;
9     end
10 end
11 exchange  $A[\ell]$  with  $A[t - 1]$ ;
12 return ( $\ell$ );
```



Expected time analysis -- intuition

- ▶ In expectation, after every constant number of recursive calls, there will be a good split,
 - ▶ **Good** split:
 - ▶ the pivot has rank in $[\frac{n}{4}, \frac{3n}{4}] \Rightarrow$ probability of a good split $p = \frac{1}{2}$
 - ▶ **Bad** split:
 - ▶ Otherwise
- ▶ Every time a good split happens,
 - ▶ the size of the problem will be reduced by at least $\frac{1}{4}$
 - ▶ i.e, the remainder size is at most $\frac{3}{4}n'$ where n' is the previous size



Expected time analysis -- intuition

- ▶ Recall $T(n) = \max(T(r-1), T(n-r)) + cn$
- ▶ Counting the cost of all good splits, we have that it is at most
 - ▶ $T_{good}(n) \leq T_{good}\left(\frac{3n}{4}\right) + cn$
 - ▶ $\Rightarrow T_{good}(n) \leq cn + \frac{3}{4}cn + \left(\frac{3}{4}\right)^2 cn + \dots = cn \left(1 + \frac{3}{4} + \left(\frac{3}{4}\right)^2 + \dots\right) = \Theta(n)$
- ▶ In-between good splits there are bad splits, but their costs intuitively can be charged to those of the good splits
 - ▶ The good split happens with probability $p = \frac{1}{2}$
 - ▶ Expected cost of bad splits is bounded by $\left(\frac{1-p}{p}\right)T_{good}(n) = T_{good}(n)$
- ▶ Hence the expected total time is $ET(n) \leq 2T_{good}(n) = \Theta(n)$



Expected time analysis -- intuition

- ▶ Recall $T(n) = \max(T(r-1), T(n-r)) + cn$
- ▶ Counting the cost of all good splits, we have that it is at most
 - ▶ $T_{good}(n) \leq T_{good}\left(\frac{3n}{4}\right) + cn$
 - ▶ $\Rightarrow T_{good}(n) \leq cn + \frac{3}{4}cn + \left(\frac{3}{4}\right)^2 cn + \dots = cn \left(1 + \frac{3}{4} + \left(\frac{3}{4}\right)^2 + \dots\right) = \Theta(n)$
- ▶ In-between good splits there are bad splits, but their costs intuitively can be charged to those of the good splits
 - ▶ The good split happens with probability $p = \frac{1}{2}$
 - ▶ Expected cost of bad splits is bounded by $\left(\frac{1-p}{p}\right)T_{good}(n) = T_{good}(n)$
- ▶ Hence the expected total time is $ET(n) \leq 2T_{good}(n) = \Theta(n)$

This is NOT a precise argument, just intuition.

This can be made more precise.

Summary

- ▶ Randomized version of QuickSelect runs in $\Theta(n)$ expected time
- ▶ In fact, one can perform Select in $\Theta(n)$ worst-case time
 - ▶ Not covered in this class.



Part E: Randomized QuickSort



Sorting revisited!

- ▶ Previously, MergeSort

- ▶ Divide and conquer paradigm
- ▶ But **NOT** in-place sorting

- ▶ Now: QuickSort

- ▶ In-place sorting
- ▶ Randomized quicksort:
 - ▶ Worst case: $\Theta(n^2)$
 - ▶ Expected running time: $\Theta(n \lg n)$



Recall MergeSort

```
MergeSort (  $A, r, s$  ) // sorting subarray  $A[r,s]$ 
```

```
if (  $r \geq s$  ) return;
```

```
 $m = (r+s) / 2$ ;
```

```
 $A1 = \text{MergeSort} ( A, r, m )$ ;
```

```
 $A2 = \text{MergeSort} ( A, m+1, s )$ ;
```

```
Merge ( $A1, A2$ );
```

- Much work has to be done in Merge(), but the “divide” step is easy (simply split the array into two equal parts).



QuickSort

```
QuickSort (  $A, r, s$  )
```

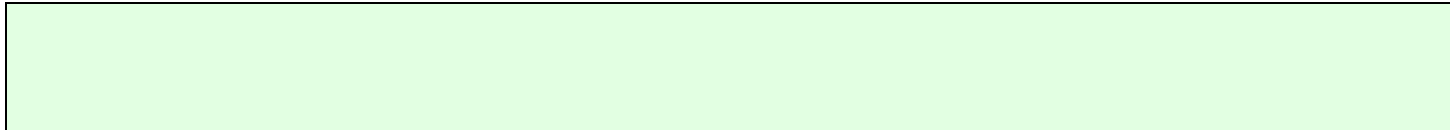
```
  if (  $r \geq s$  ) return;
```

```
   $m$  = Partition (  $A, r, s$  );
```

```
   $A1$  = QuickSort (  $A, r, m$  );
```

```
   $A2$  = QuickSort (  $A, m+1, s$  );
```

```
  Merge ( $A1, A2$ );
```



$A[m]$: pivot

QuickSort

QuickSort (A, r, s)

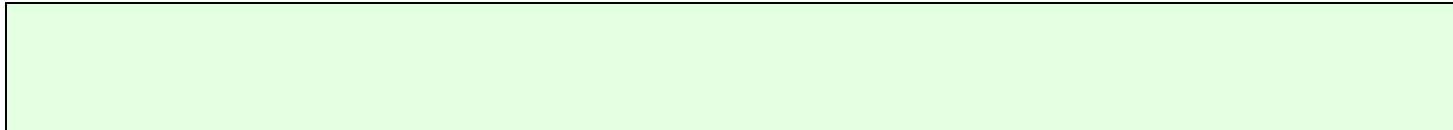
if ($r \geq s$) return;

m = Partition (A, r, s);

$A1$ = QuickSort (A, r, m);

$A2$ = QuickSort ($A, m+1, s$);

~~Merge ($A1, A2$);~~



$A[m]$: pivot

QuickSort

```
QuickSort (  $A, r, s$  )
```

```
  if (  $r \geq s$  ) return;
```

```
   $m$  = Partition (  $A, r, s$  );
```

```
   $A1$  = QuickSort (  $A, r, m-1$  );
```

```
   $A2$  = QuickSort (  $A, m+1, s$  );
```

► Worst case

► Best case



QuickSort

QuickSort (A, r, s)

if ($r \geq s$) return;

m = Partition (A, r, s);

$A1$ = QuickSort ($A, r, m-1$);

$A2$ = QuickSort ($A, m+1, s$);

► **Worst case**

► $T(n) = T(n-1) + cn = \Theta(n^2)$

► **Best case**

► $T(n) = 2T\left(\frac{n}{2}\right) + cn = \Theta(n \lg n)$



rand-QuickSort

```
rand-QuickSort (  $A, r, s$  )
```

```
  if (  $r \geq s$  ) return;
```

```
   $m$  = rand-Partition (  $A, r, s$  );
```

```
   $A1$  = rand-QuickSort (  $A, r, m-1$  );
```

```
   $A2$  = rand-QuickSort (  $A, m+1, s$  );
```



rand-QuickSort

```
rand-QuickSort (  $A, r, s$  )
```

```
if (  $r \geq s$  ) return;
```

```
 $m$  = rand-Partition (  $A, r, s$  );
```

```
 $A1$  = rand-QuickSort (  $A, r, m-1$  );
```

```
 $A2$  = rand-QuickSort (  $A, m+1, s$  );
```

- ▶ **Worst case**

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

- ▶ **Best case**

- ▶ $T(n) = 2T\left(\frac{n}{2}\right) + cn = \Theta(n \lg n)$



▶ **rand-QuickSelect**

- ▶ like rand-Select, there are good and bad splits
- ▶ as long as good splits come constant fraction of the time, the time complexity is dominated by good splits
- ▶ expected running time is $ET(n) = \Theta(n \lg n)$



▶ rand-QuickSelect

- ▶ like rand-Select, there are good and bad splits
- ▶ as long as good splits come constant fraction of the time, the time complexity is dominated by good splits
- ▶ expected running time is $ET(n) = \Theta(n \lg n)$

▶ Compared to MergeSort

- ▶ In-place sorting
 - ▶ while MergeSort needs to open a new output array of size $\Theta(n)$
- ▶ In practice often faster, and needs much smaller memory (important!)



FIN

