

ENPM 809X

E. Gunduzhan

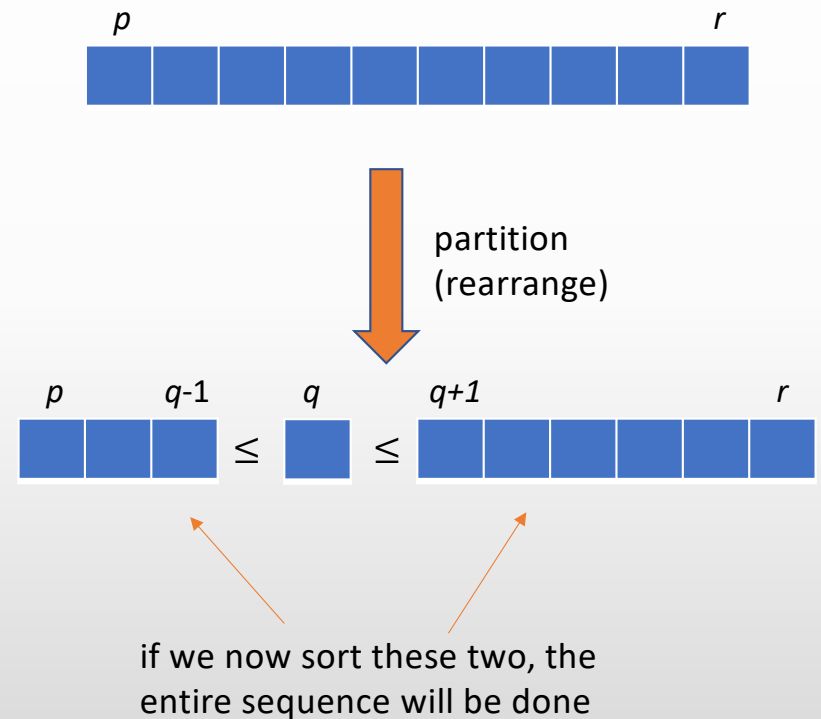
Module 4: Quicksort; Sorting in Linear Time

Worst Case vs Average Running Time

- Worst-case running time of quicksort is $\Theta(n^2)$ – worse than heapsort and merge sort.
- However, it is very efficient on average – its expected running time is $\Theta(n \lg n)$, with small constant factors.
- This usually makes quicksort the best practical choice for sorting.

Algorithm

- Quicksort works by recursion, similar to merge sort
- Divide: Partition given numbers $A[p..r]$ into three subsequences $A[p..q-1]$, $A[q]$, $A[q+1..r]$ such that
 - Each element of $A[p..q-1]$ is less than or equal to $A[q]$, and
 - $A[q]$ is less than or equal to each element of $A[q+1..r]$
- Conquer: Sort two subsequences recursively
- Combine: No additional action – the entire sequence is now sorted



Partitioning

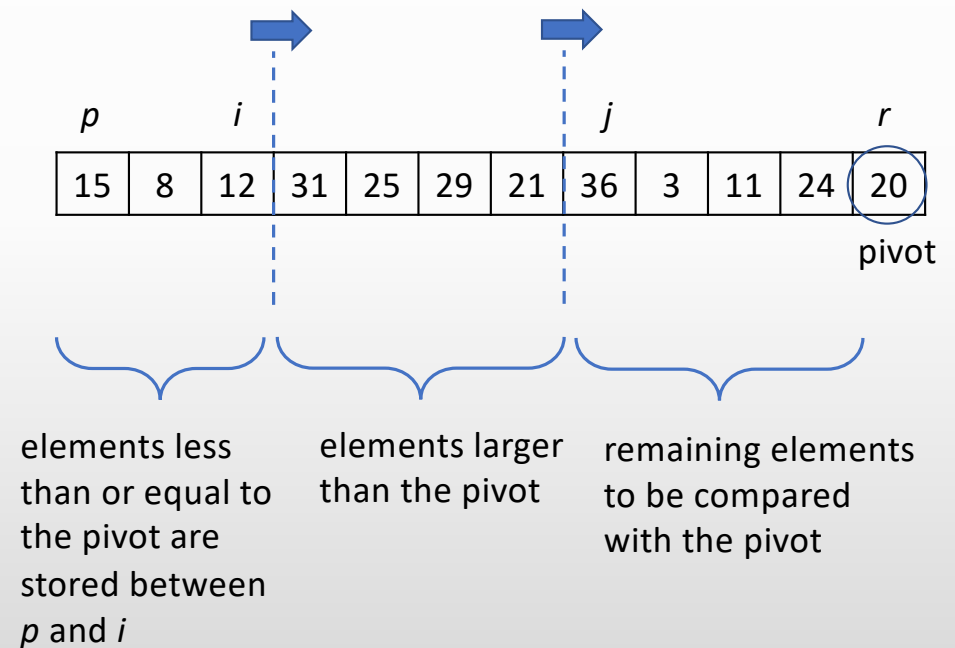
Partition(A, p, r)

Choose the last element as the pivot for partitioning

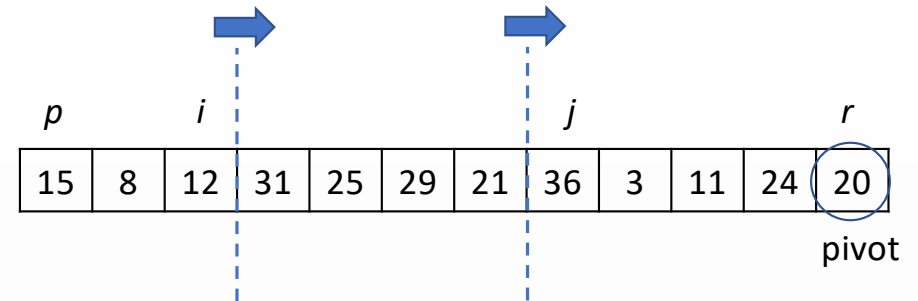
Move each element less than or equal to x to the first partition (initially empty)

Place the pivot just outside the boundary of the first partition

Return index of the pivot



Partitioning



Partition(A, p, r)

$x = A[r]$ → Choose the last element as the pivot for partitioning

$i = p - 1$ → Initialize i – elements less than or equal to x will be stored between p and i

for $j = p$ to $r - 1$

 if $A[j] \leq x$

$i = i + 1$

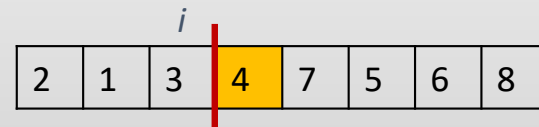
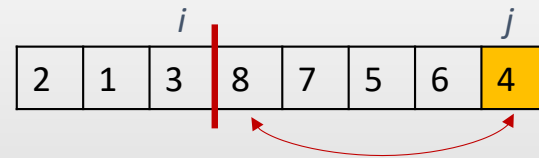
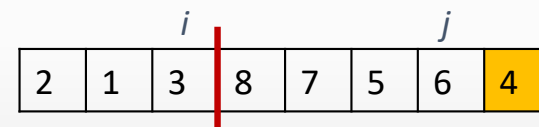
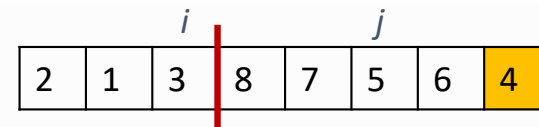
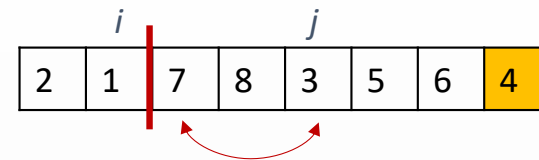
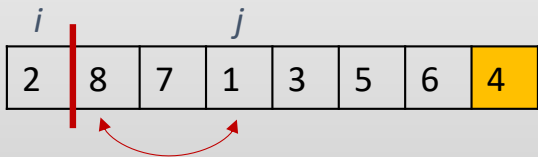
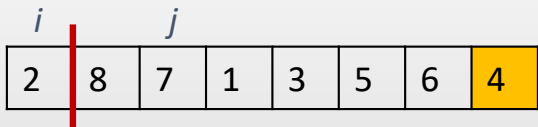
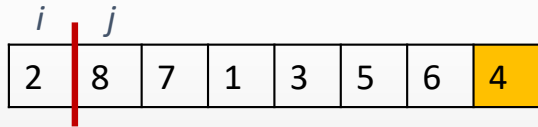
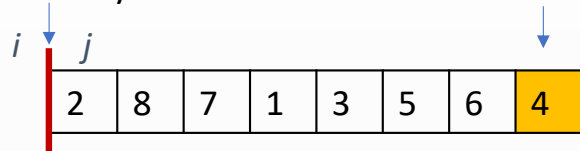
 swap $A[i]$ and $A[j]$ } For each element less than or equal to x , expand the first partition and move the element to that partition

swap $A[i+1]$ and $A[r]$ } Move the pivot just outside boundary of the first partition and return its index

return $i + 1$

Example

partition boundary pivot



Example

13	19	9	5	12	8	17	14	21	2	16	11
----	----	---	---	----	---	----	----	----	---	----	----

i	j	13	19	9	5	12	8	17	14	21	2	16	11
-----	-----	----	----	---	---	----	---	----	----	----	---	----	----

i	j	13	19	9	5	12	8	17	14	21	2	16	11
-----	-----	----	----	---	---	----	---	----	----	----	---	----	----

i	j	9	19	13	5	12	8	17	14	21	2	16	11
-----	-----	---	----	----	---	----	---	----	----	----	---	----	----

i	j	9	5	13	19	12	8	17	14	21	2	16	11
-----	-----	---	---	----	----	----	---	----	----	----	---	----	----

	i		j	9	5	13	19	12	8	17	14	21	2	16	11
--	-----	--	-----	---	---	----	----	----	---	----	----	----	---	----	----

	i		j	9	5	8	19	12	13	17	14	21	2	16	11
--	-----	--	-----	---	---	---	----	----	----	----	----	----	---	----	----

	i		j	9	5	8	19	12	13	17	14	21	2	16	11
--	-----	--	-----	---	---	---	----	----	----	----	----	----	---	----	----

	i		j	9	5	8	2	12	13	17	14	21	19	16	11
--	-----	--	-----	---	---	---	---	----	----	----	----	----	----	----	----

	i		j	9	5	8	2	12	13	17	14	21	19	16	11
--	-----	--	-----	---	---	---	---	----	----	----	----	----	----	----	----

	i		j	9	5	8	2	11	13	17	14	21	19	16	12
--	-----	--	-----	---	---	---	---	----	----	----	----	----	----	----	----

Correctness of Partitioning Algorithm

Partition(A, p, r)

$x = A[r]$

$i = p - 1$

for $j = p$ to $r - 1$

 if $A[j] \leq x$

$i = i + 1$

 swap $A[i]$ and $A[j]$

swap $A[i+1]$ and $A[r]$

return $i + 1$

- Loop invariant:
 - $A[k] \leq x$ for any k such that $p \leq k \leq i$, and
 - $A[k] > x$ for any k such that $i + 1 \leq k \leq j - 1$
- Initialization:

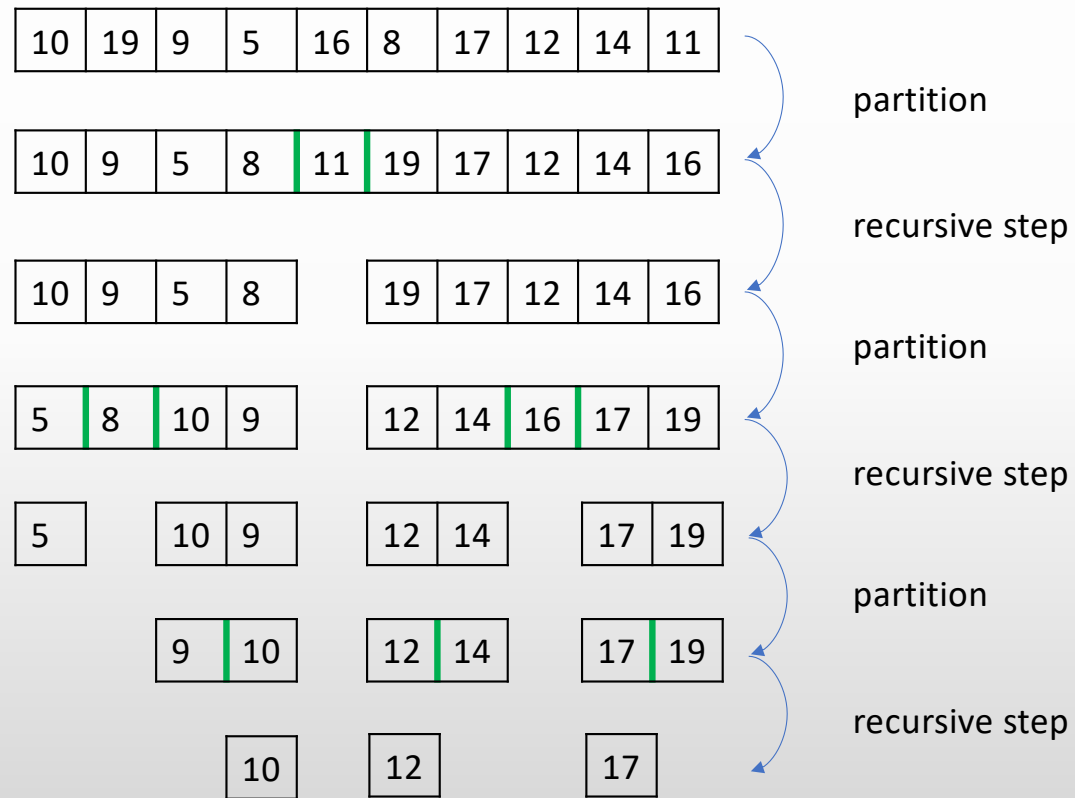
Trivially true since both sets are initially empty
- Maintenance:

If $A[j] > x$, only action is incrementing j . First set doesn't change, and an element larger than x is added to the second set.

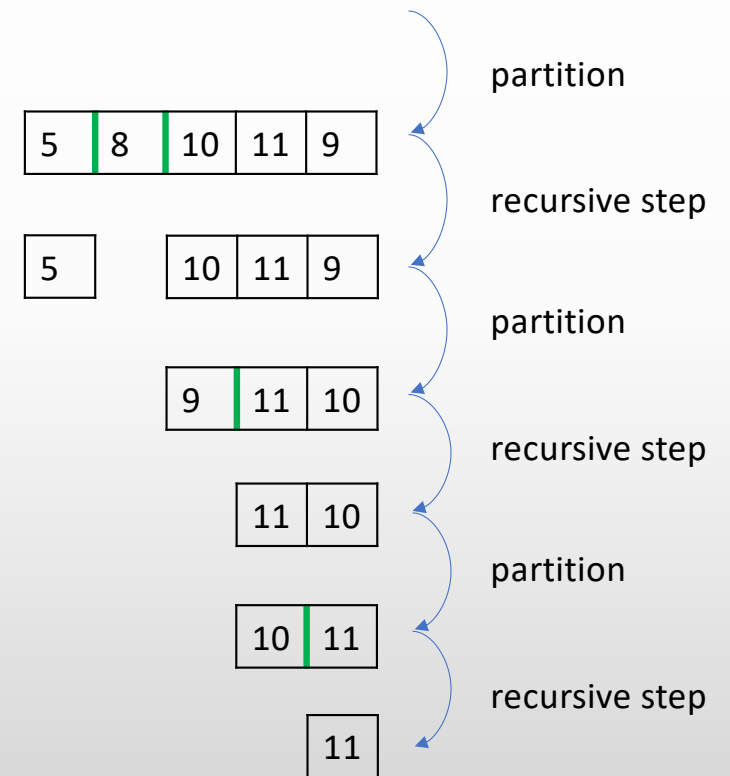
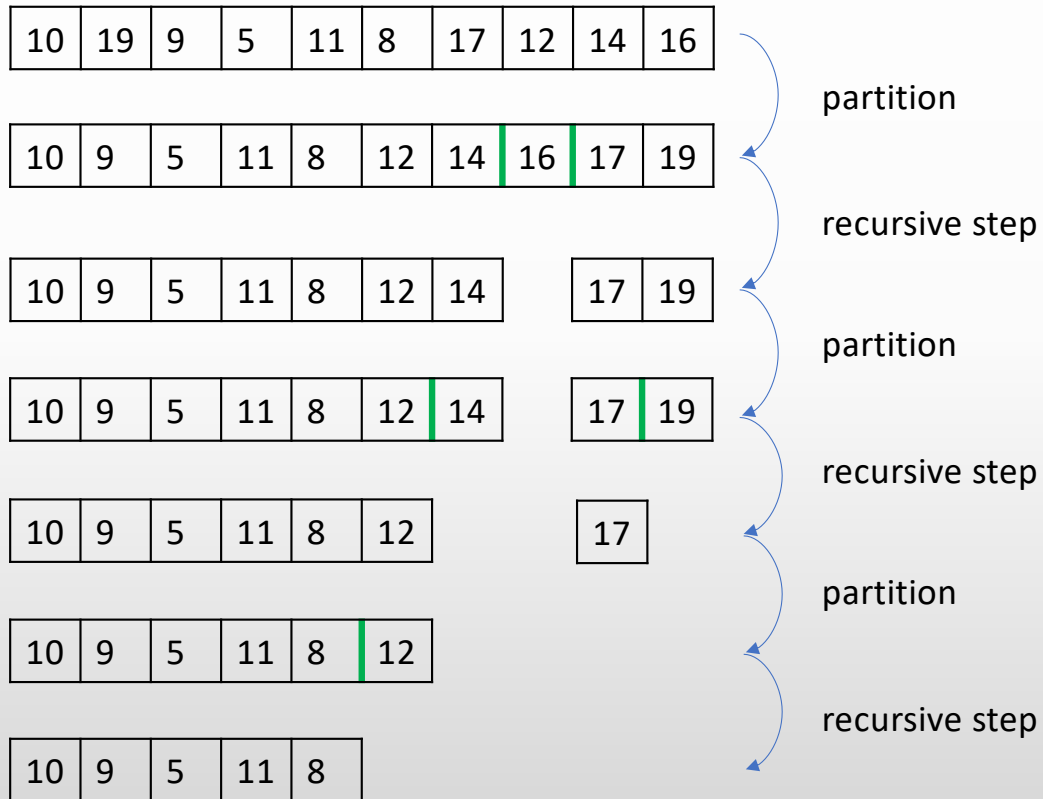
If $A[j] \leq x$, this entry is moved to the first set and replaced by $A[i + 1]$ which is larger than x .
- Termination:

At the end we swap x with $A[i + 1]$ which is larger than x . So, entire array is correctly partitioned.

Balanced Partitioning



Unbalanced Partitioning



Quicksort and its Running Time

Quicksort(A, p, r)

if $p < r$

$q = \text{Partition}(A, p, r)$

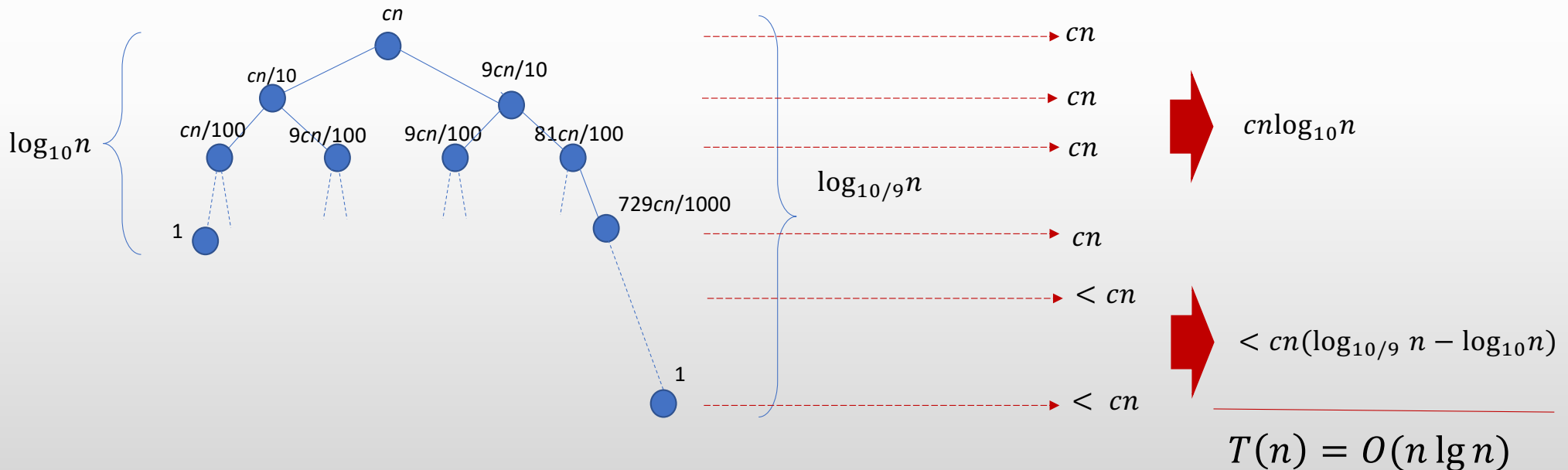
 Quicksort($A, p, q-1$)

 Quicksort($A, q+1, r$)

- Partition algorithm runs in $\Theta(n)$, where $n = r - p + 1$.
- Running time of quicksort depends on how balanced the partitioning is.
- If not well-balanced, then running time can be as large as insertion sort
 - Worst case occurs when one partition has $n - 1$ elements and the other is empty
 - $T(n) = T(n - 1) + \Theta(n) \Rightarrow T(n) = \Theta(n^2)$
- If well-balanced, then running time is as low as the merge sort
 - Best case occurs when two partitions have $\sim n/2$ elements
 - $T(n) = 2T(n/2) + \Theta(n) \Rightarrow T(n) = \Theta(n \lg n)$

How Bad are “Bad Partitions”?

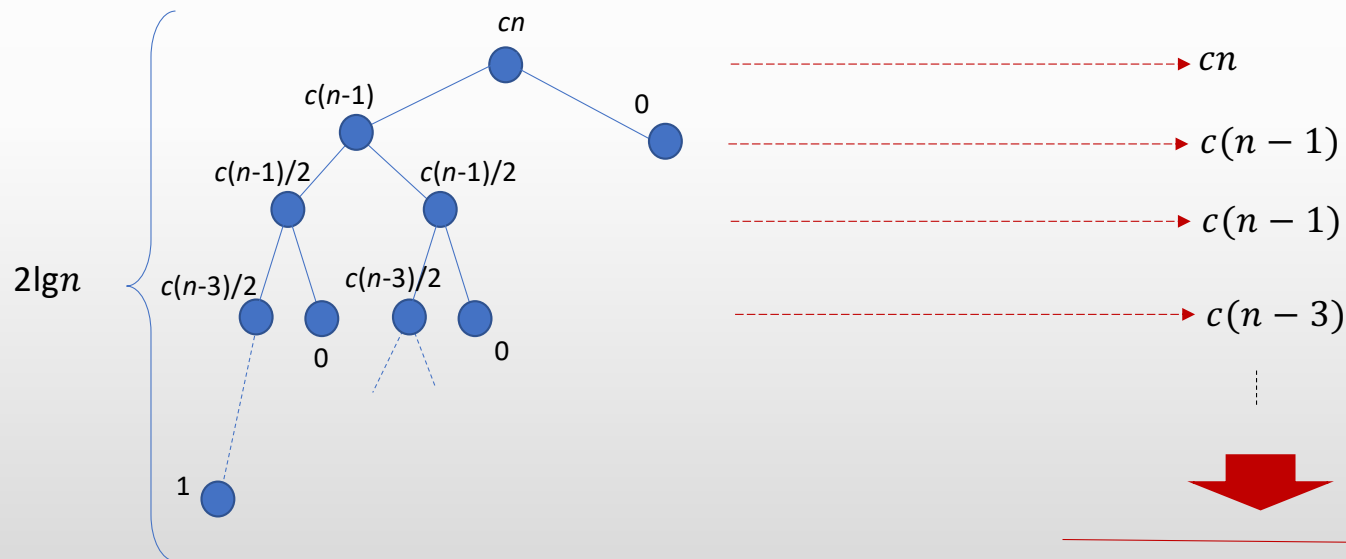
- Partitioning with any constant proportion still yields $O(n \lg n)$ running time.
 - Total processing time at each level is cn , and there are $\Theta(\lg n)$ levels $\Rightarrow O(n \lg n)$
- Example (1/10 – 9/10 Split): $T(n) = T(n/10) + T(9n/10) + cn$



- So, worst case occurs when partitioning is not constant proportion

Analysis of the Average Running Time

- If good and bad splits alternated, the running time would still be $O(n \lg n)$



$$T(n) = O(n \lg n)$$

Analysis of the Average Running Time (cont'd)

- We can randomize the partitioning algorithm by selecting the pivot randomly over the entire input array, instead of the last element
- The running time of quicksort strictly depends on the total number of times the comparison inside the for loop is executed
- Calculating an upper bound on the expected number of comparisons gives an upper bound on the running time of quicksort

Randomized-Partition(A, p, r)

```
 $i = \text{Random}(p, r)$   
swap  $A[i]$  and  $A[r]$   
return Partition( $A, p, r$ )
```

Partition(A, p, r)

```
 $x = A[r]$   
 $i = p - 1$   
for  $j = p$  to  $r - 1$   
    if  $A[j] \leq x$   
         $i = i + 1$   
        swap  $A[i]$  and  $A[j]$   
swap  $A[i+1]$  and  $A[r]$   
return  $i + 1$ 
```

Analysis of the Average Running Time (cont'd)

- Let $z_1 \leq z_2 \leq \dots \leq z_n$ be the elements of A arranged in increasing order.
- Observation: Any two elements, z_i and z_j , are compared at most once. (why?)



- Define $X_{ij} = I\{z_i \text{ is compared to } z_j\}$
- Then, the expected number of comparisons is

$$E[X] = \sum_{i=1}^{n-1} \sum_{j=i+1}^n E[X_{ij}] = \sum_{i=1}^{n-1} \sum_{j=i+1}^n \Pr\{z_i \text{ is compared to } z_j\}$$

Analysis of the Average Running Time (cont'd)

- Let $Z_{ij} = \{z_i, \dots, z_j\}$ be the set of elements between z_i and z_j .
- Observation: If one of the elements $z_{i+1}, z_{i+2}, \dots, z_{j-1}$ is selected as a pivot before z_i or z_j , then z_i and z_j are never compared with each other; otherwise they will be compared with each other.
- If the pivot selection is randomized, then

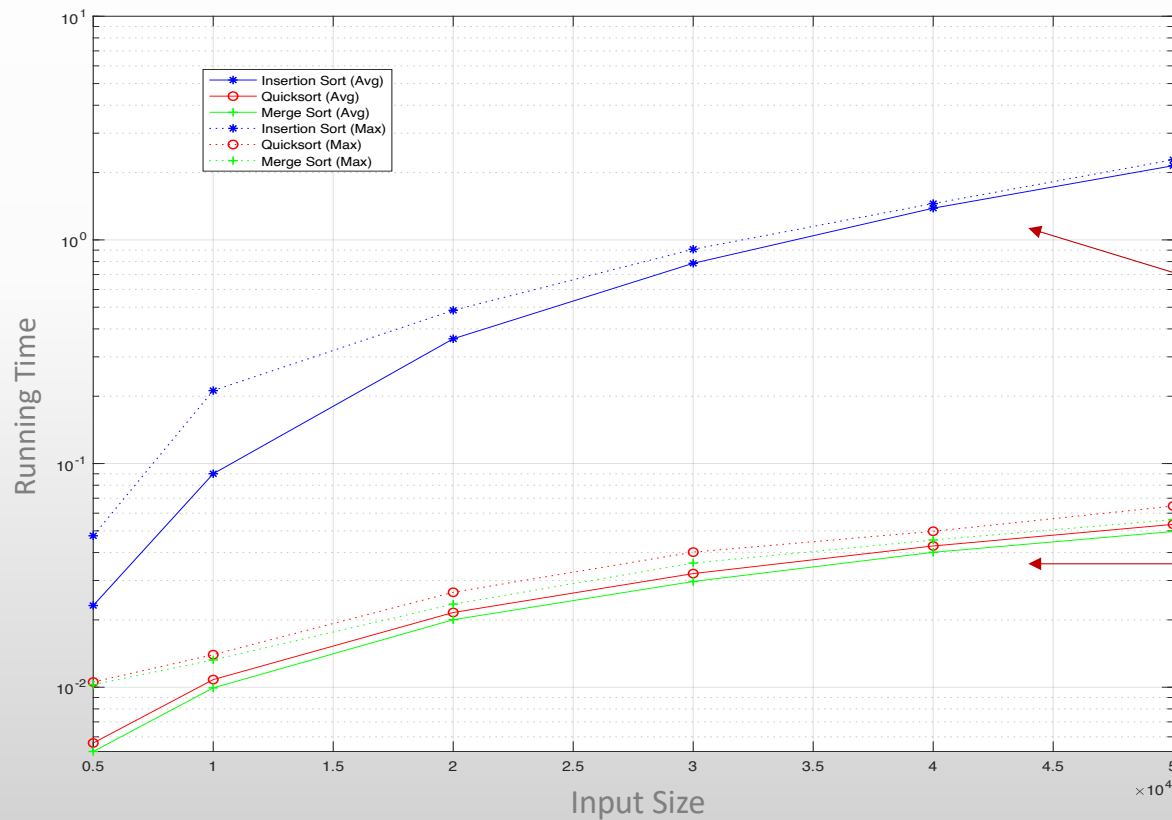
$$\Pr\{z_i \text{ is compared to } z_j\} = 2/(j - i + 1)$$

- Therefore,

$$E[X] = \sum_{i=1}^{n-1} \sum_{j=i+1}^n \frac{2}{j - i + 1} < \sum_{i=1}^{n-1} \sum_{k=1}^n \frac{2}{k} = O(n \lg n)$$

- Since the best case running time is $\Theta(n \lg n)$, it follows that above bound on expected running time is tight, i.e. $E[X] = \Theta(n \lg n)$.

Numerical Comparison of Insertion Sort, Quicksort, Merge Sort



Average and maximum running times over 100 random inputs

Insertion sort has largest running times

Quicksort performs very closely to merge sort

Sorting in Linear Time

Lower Bound on Comparison Sorts

- All sorting algorithms we have studied so far are comparison sorts.
- In a comparison sort, the only operation between elements is a comparison, i.e. $a \leq b$, $a > b$, $a = b$, etc.
- Any comparison sort algorithm requires $\Omega(n \lg n)$ in the worst case.
 - Therefore merge sort and heapsort are asymptotically optimal
- There are other sorting algorithms (i.e. not comparison), that require only $\Theta(n)$ running time under certain assumptions on the input.

Counting Sort

- Assumption: Each of the n elements in the input A should be an integer in the range $1..k$, where k is an integer.
- If $k = O(n)$, then counting sort runs in $\Theta(n)$ time.
- Instead of element-wise comparisons, counting sort first determines the number of elements in A less than or equal to j , for all $j = 1, \dots, k$; and
- Then uses this information to find sorted positions at the output.
 - Example: If there are 16 elements in A that are less than x , which position in the sorted output will x be?
- The algorithm uses two other arrays, B and C , in addition to the input array A .
- Counting sort is *stable*, i.e. it doesn't change the relative order of equal elements.

Counting Sort (cont'd)

Counting-Sort(A, B, k)

init $C[1..k]$ to all zeros

for $j = 1$ to $A.length$

$C[A[j]] = C[A[j]] + 1$



Record number of occurrences in C , i.e. if number p occurs m times in A then $C[p] = m$

for $j = 2$ to k

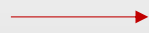
$C[j] = C[j] + C[j-1]$



Convert to cumulative occurrences, i.e. if numbers $\leq p$ occur r times in A then $C[p] = r$

for $j = A.length$ downto 1

$B[C[A[j]]] = A[j]$



$C[A[j]]$ shows how many numbers $\leq A[j]$ are in A .

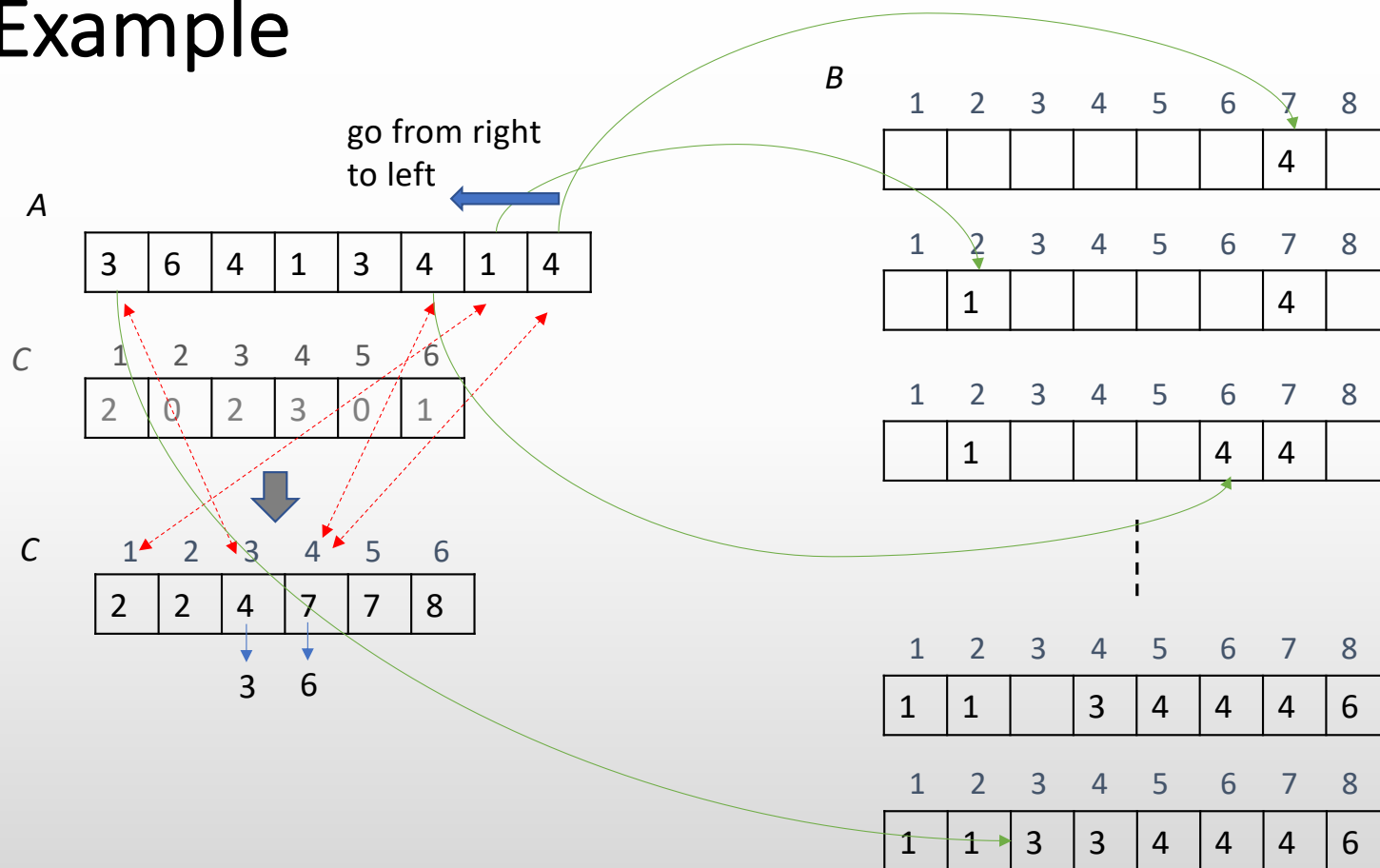
$C[A[j]] = C[A[j]] - 1$



Therefore, $A[j]$ should be at $C[A[j]]^{\text{th}}$ element of the output array B .

There may be other elements in A equal to $A[j]$. We decrement the count so that these go to adjacent positions in B .

Example



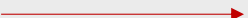
Running Time of Counting Sort

Counting-Sort(A, B, k)

init $C[1..k]$ to all zeros

for $j = 1$ to $A.length$
 $C[A[j]] = C[A[j]] + 1$  Takes $\Theta(n)$

for $j = 2$ to k
 $C[j] = C[j] + C[j-1]$  Takes $\Theta(k)$

for $j = A.length$ downto 1
 $B[C[A[j]]] = A[j]$  Takes $\Theta(n)$
 $C[A[j]] = C[A[j]] - 1$

Therefore the running time is $\Theta(k + n)$. When $k = O(n)$, this becomes $\Theta(n)$.

Intuitively, if $k \gg n$, then using counting sort will not be efficient.

Radix Sort

- Suppose we need to sort some records according to multiple fields, (e.g. year, month, day).
- We can use a comparison sort to compare elements using multiple fields, or...
- We can first sort according to the first field, then sort according to the second field, ... and so on.
- Radix sort, counterintuitively, uses the least significant field in the first sort, then the second least significant, ... and sorts lastly using the most significant field.

Radix Sort (cont'd)

- Suppose the records to be sorted are d -digit numbers.
- If d is large, using counting sort may be inefficient.
- Radix sort uses a “stable” sorting algorithm, such as counting sort, and first sorts using the first digit, then the second digit, ... etc.
 - There can at most be 10 different values during each sort, so counting sort would be efficient

Radix-Sort(A, d)

for $i = 1$ to d

sort A using digit i

329	720	720	329
457	355	329	355
657	436	436	436
839	457	839	457
436	657	355	657
720	329	457	720
355	839	657	839

Correctness and Running Time of Radix Sort

- Correctness can be shown by induction. Assume we are sorting d -digit numbers...
 - For any $i = 1, \dots, d$, if the numbers are already sorted according to their least significant $(i-1)$ digits, then they will be sorted according to their least significant i digits after i th step.
 - The sort at i th step (e.g. counting sort) puts the i th digits into correct order.
 - Since it is a stable algorithm, relative order of numbers with the same i th digit does not change
- If the sort algorithm at each digit takes $\Theta(n + k)$ time, then radix sort takes $\Theta(d(n + k))$ time.

Bucket Sort

- Assumption: The input is uniformly distributed in the interval $[0,1)$. This means, for each element of the input, $0 \leq A[i] < 1$.
- Bucket sort operates as follows:
 - Divides the interval $[0,1)$ into n equally-sized intervals (“buckets”)
 - Distributes the input elements into the buckets
 - Sorts each bucket
 - Combines the buckets
- Uses an auxiliary array B of linked lists

Bucket Sort (cont'd)

Bucket-Sort(A)

init $B[0..n-1]$ as an empty list

$n = A.length$

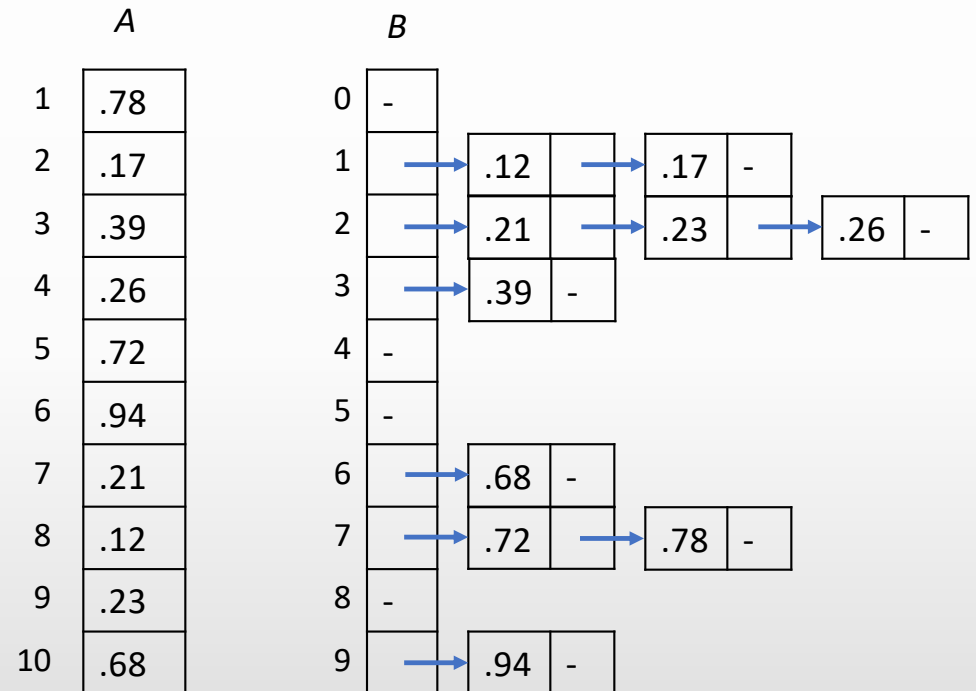
for $i = 1$ to n

 insert $A[i]$ into $B[\lfloor nA[i] \rfloor]$

for $i = 1$ to n

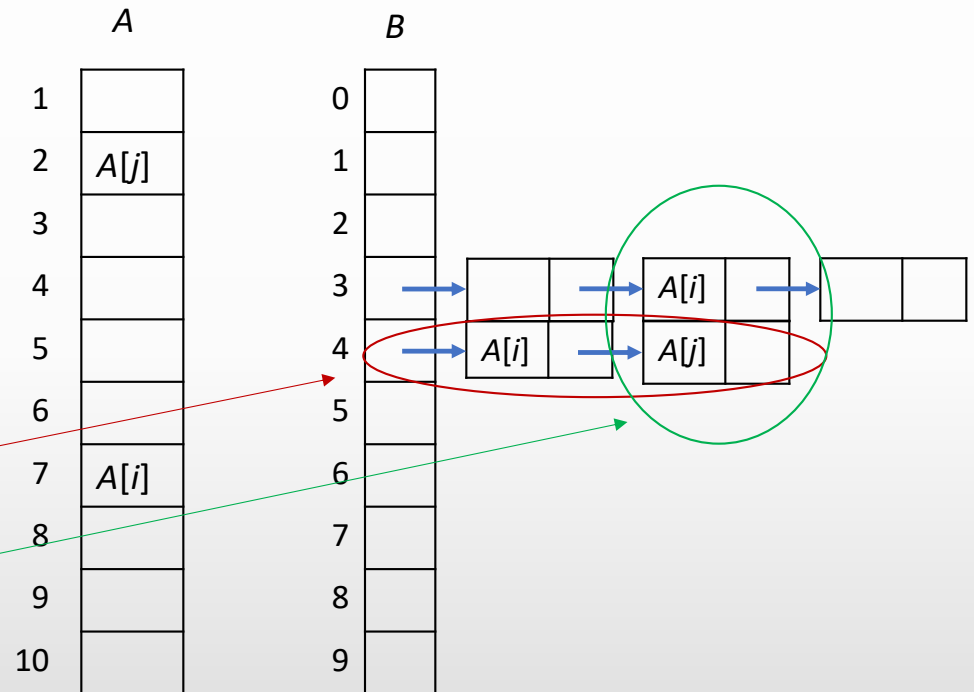
 sort list $B[i]$

concatenate all lists together



Correctness

- Consider any two elements $A[i]$ and $A[j]$.
- Assume $A[i] \leq A[j]$.
- Then, for corresponding buckets in B , we have $\lfloor nA[i] \rfloor \leq \lfloor nA[j] \rfloor$.
- If equality, they go into same bucket
- If strict inequality, $A[i]$ goes to a lower bucket than $A[j]$.
- Since each bucket is sorted and then combined in increasing order, they will be in correct order in the output.



Average Running Time

Bucket-Sort(A)

init $B[0..n-1]$ as an empty list

$n = A.length$

for $i = 1$ to n

 insert $A[i]$ into $B[\lfloor nA[i] \rfloor]$

for $i = 1$ to n

 sort list $B[i]$

concatenate all lists together

$\Theta(n)$

Sort lists of random size

Since n elements are distributed to n lists, intuitively the number of elements in each list must be small

Average Running Time (cont'd)

- If k_i is the number of elements in list i , then

$$E\{T(n)\} = E\{\Theta(n) + \sum_{i=0}^{n-1} O(k_i^2)\} = \Theta(n) + \sum_{i=0}^{n-1} O(E\{k_i^2\})$$

- $E\{k_i^2\}$ does not depend on i , and it can be shown that (see the book):

$$E\{k_i^2\} = 2 - 1/n$$

- Therefore average running time is asymptotically linear in n

$$E\{T(n)\} = \Theta(n) + nO(2 - 1/n) = \Theta(n)$$

- What is the worst-case running time?