Analysis and Design of Algorithms

Sorting (Part B: Divide and Conquer)

Instructor: Morteza Zakeri

Recap: Sorting incremental

- Insertion sort
  - Design approach: incremental
  - Sorts in place: Yes
  - Best case: $\Theta(n)$
  - Worst and average case: $\Theta(n^2)$

- Bubble sort
  - Design approach: incremental
  - Sorts in place: Yes
  - Running time: $\Theta(n^2)$

Recap: Sorting incremental

- Selection sort
  - Design approach: incremental
  - Sorts in place: Yes
  - Running time: $\Theta(n^2)$

New: Sorting with Divide and Conquer

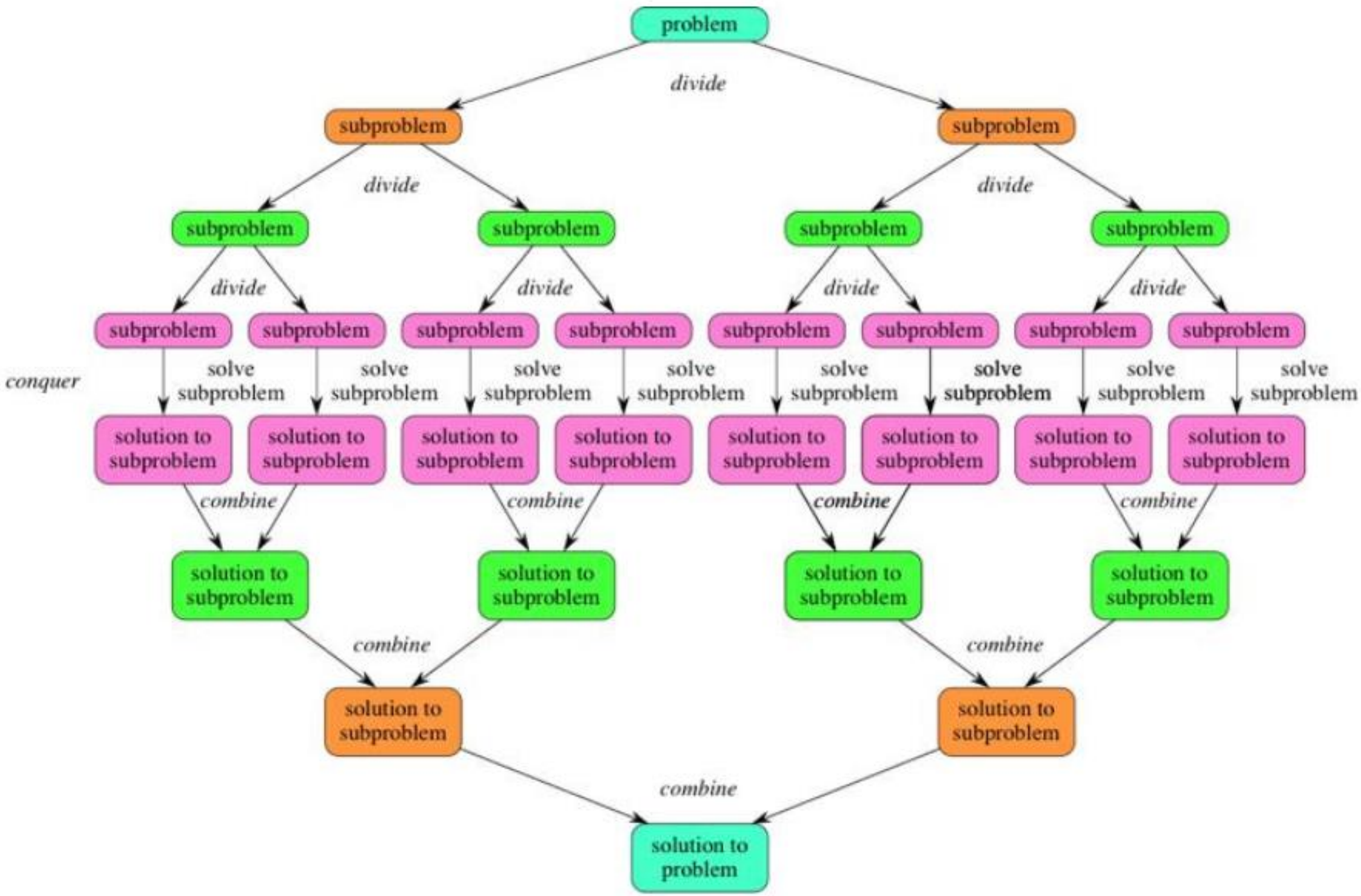
- **Merge Sort**
  - Design approach: **divide and conquer**
  - Sorts in place: No
  - Running time: Let's see!!

- **Quick Sort**
  - Design approach: **divide and conquer**
  - Sorts in place: Yes
  - Running time: Let's see!!

Recap: Divide-and-Conquer

- **Divide** the problem into a number of sub-problems
  - Similar sub-problems of smaller size
- **Conquer** the sub-problems
  - Solve the sub-problems **recursively**
  - Sub-problem size small enough $\Rightarrow$ solve the problems in straightforward manner
- **Combine** the solutions of the sub-problems
  - Obtain the solution for the original problem

Recap: Divide-and-Conquer sketch



Merge Sort Approach

- To sort an array $A[p\ ..\ r]$:
- **Divide**
  - Divide the n-element sequence to be sorted into two subsequences of $n/2$ elements each
- **Conquer**
  - Sort the subsequences recursively using merge sort
  - When the size of the sequences is 1 there is nothing more to do
- **Combine**
  - Merge the two **sorted subsequences**

Merge Sort

p			q				r
1	2	3	4	5	6	7	8
5	2	4	7	1	3	2	6

Alg.: MERGE-SORT(A, p, r)

if p < r ▷ Check for base case

then $q \leftarrow \lfloor (p + r)/2 \rfloor$ ▷ Divide

MERGE-SORT(A, p, q) ▷ Conquer

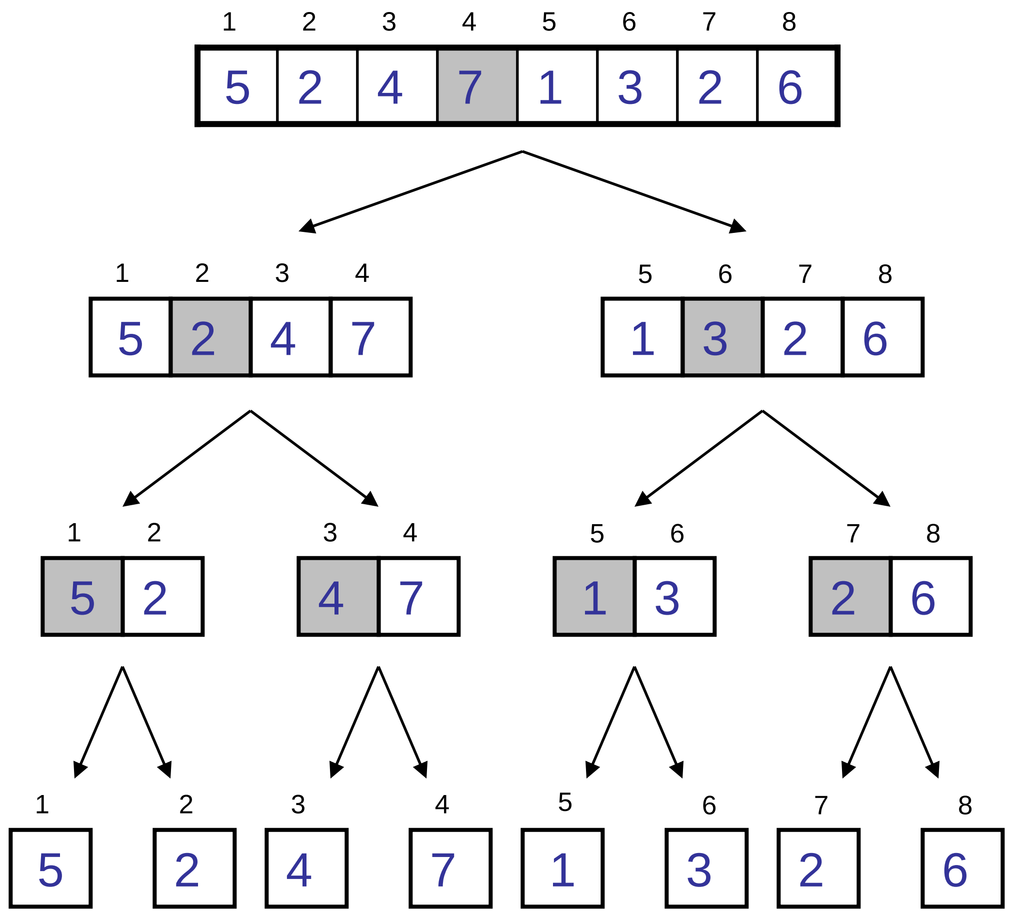
MERGE-SORT(A, q + 1, r) ▷ Conquer

MERGE(A, p, q, r) ▷ Combine

- Initial call: MERGE-SORT(A, 1, n)

Example – n Power of 2

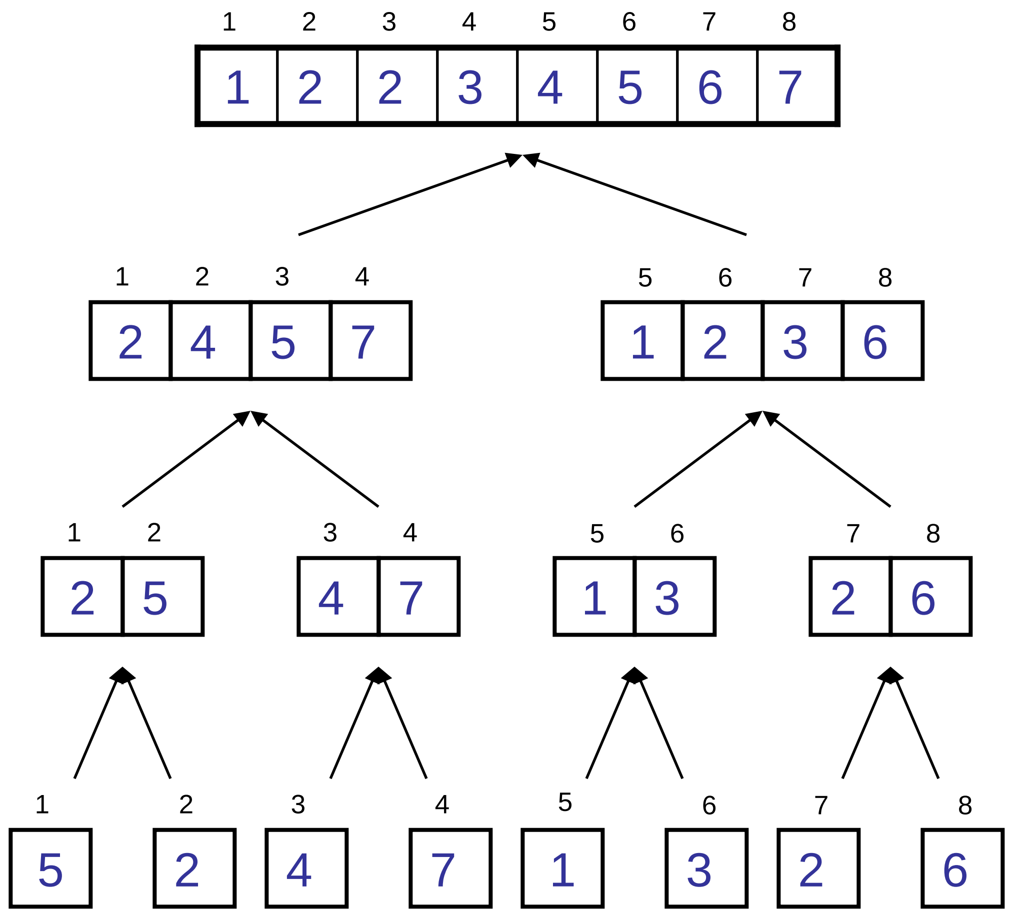
Divide



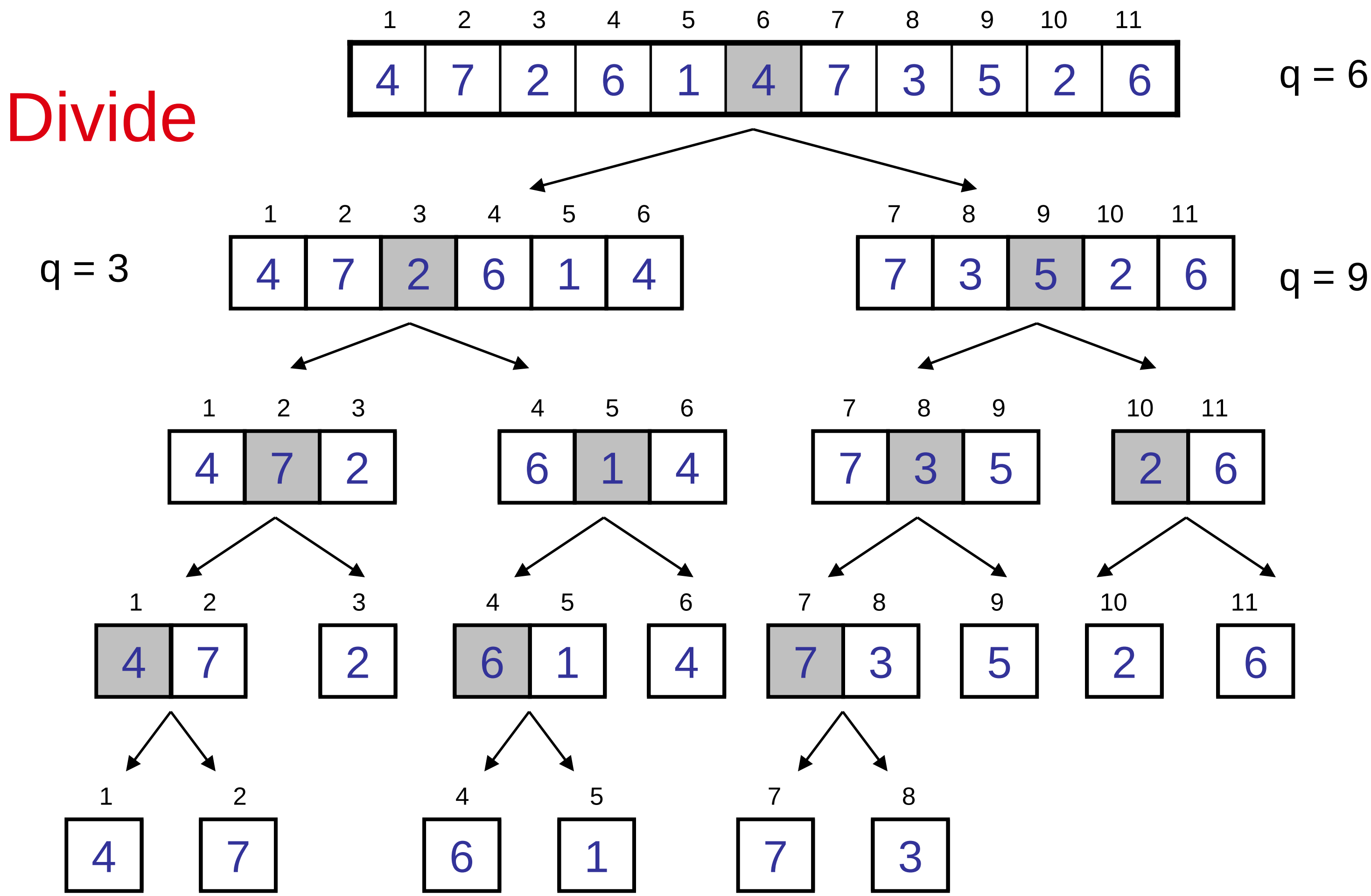
q = 4

Example – n Power of 2

Conquer
and
Merge

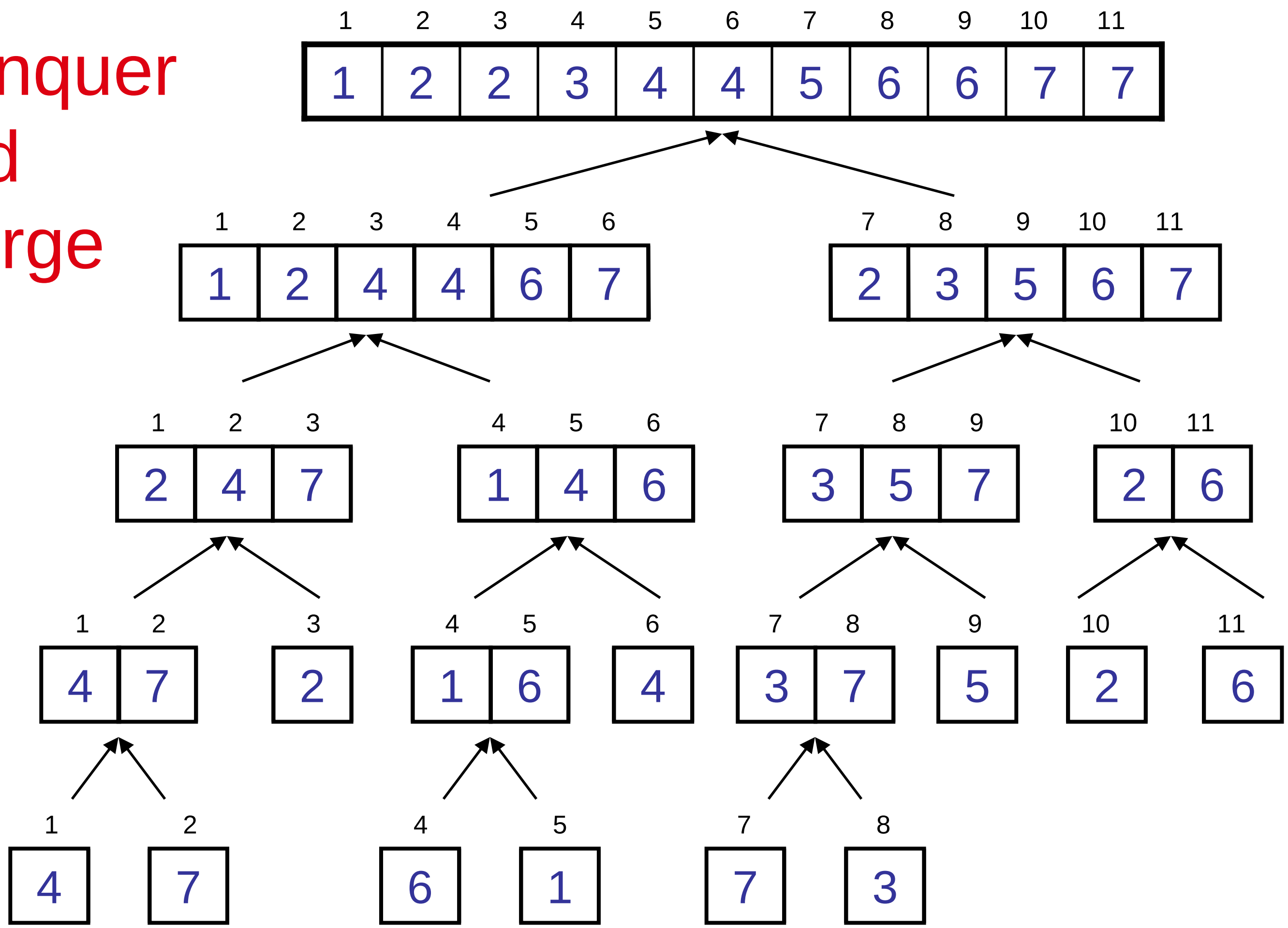


Example – n Not a Power of 2



Example – *n* Not a Power of 2

Conquer and Merge

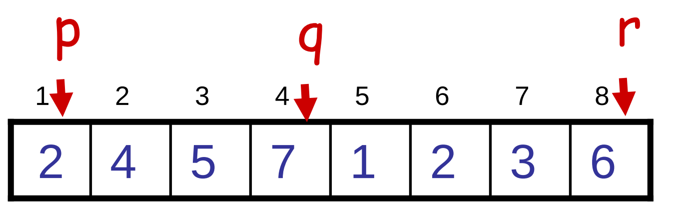


Merging

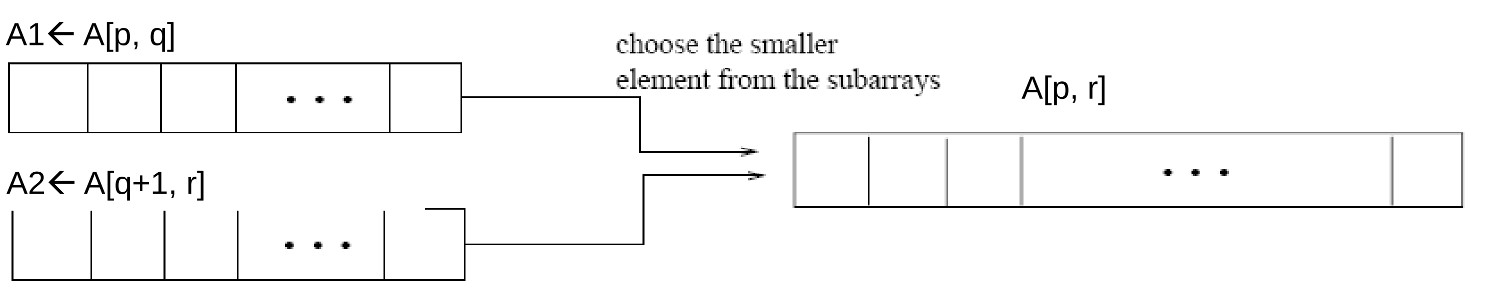
p			q				r
1	2	3	4	5	6	7	8
2	4	5	7	1	2	3	6

- **Input:** Array A and indices p, q, r such that $p \le q < r$
  - Subarrays $A[p \,..\, q]$ and $A[q + 1 \,..\, r]$ are sorted
- **Output:** One single sorted subarray $A[p \,..\, r]$

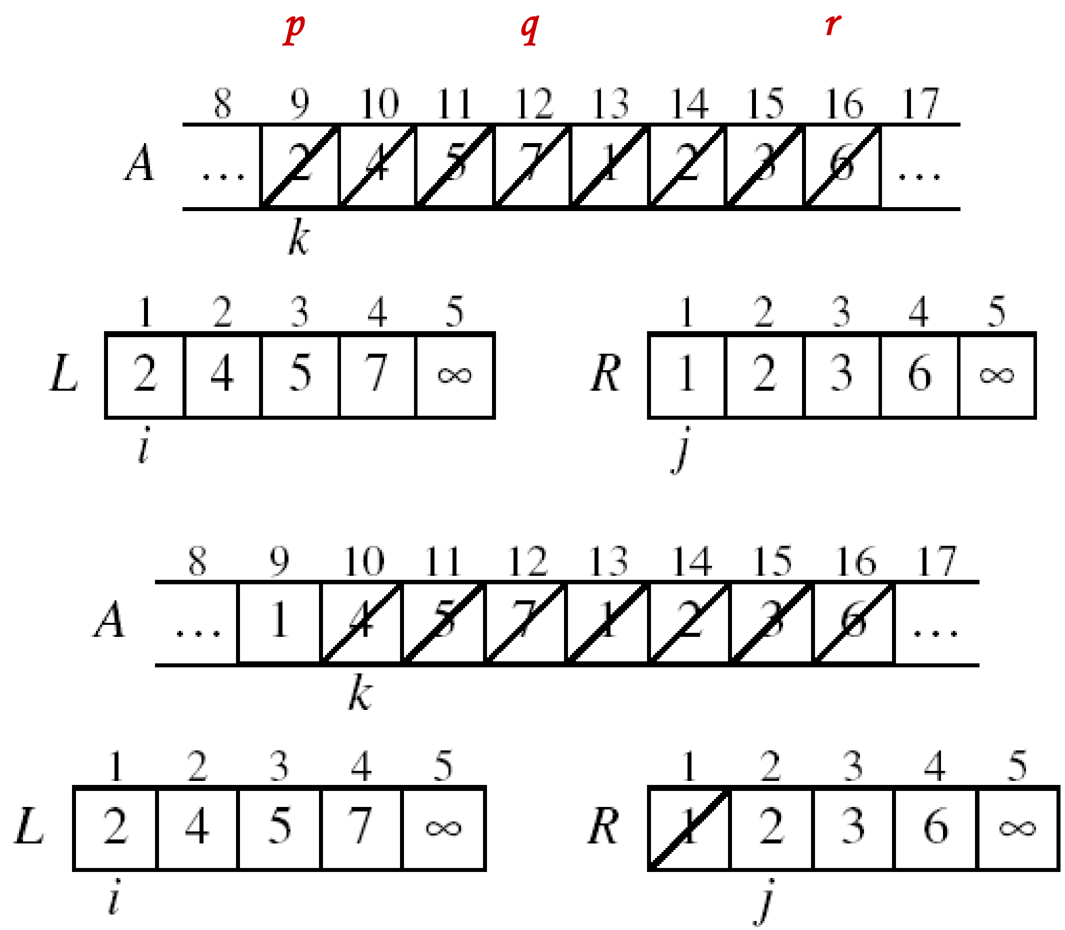
Merging



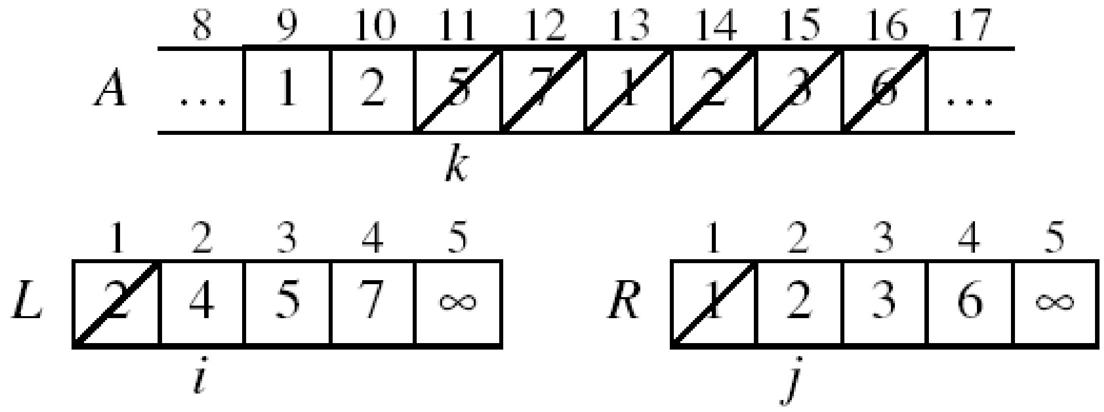
- Idea for merging:
  - Two piles of sorted cards
    - Choose the smaller of the two top cards
    - Remove it and place it in the output pile
  - Repeat the process until one pile is empty
  - Take the remaining input pile and place it face-down onto the output pile

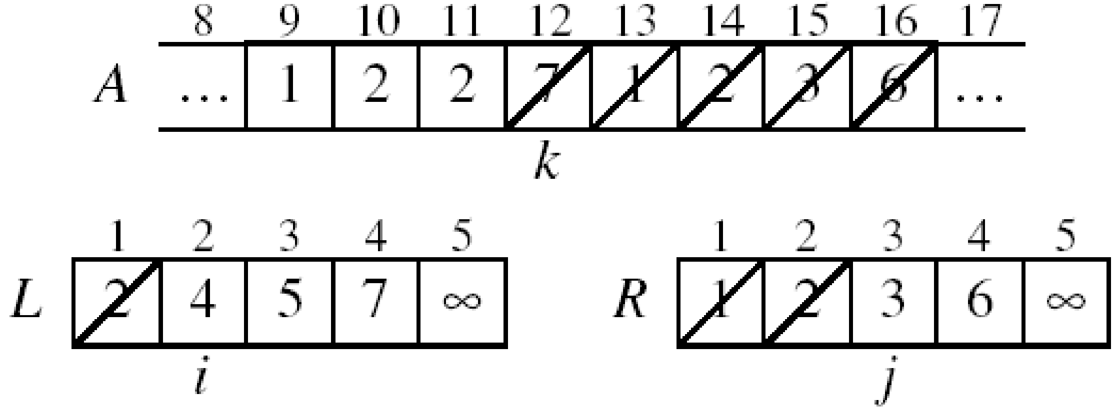


Example: MERGE(A, 9, 12, 16)

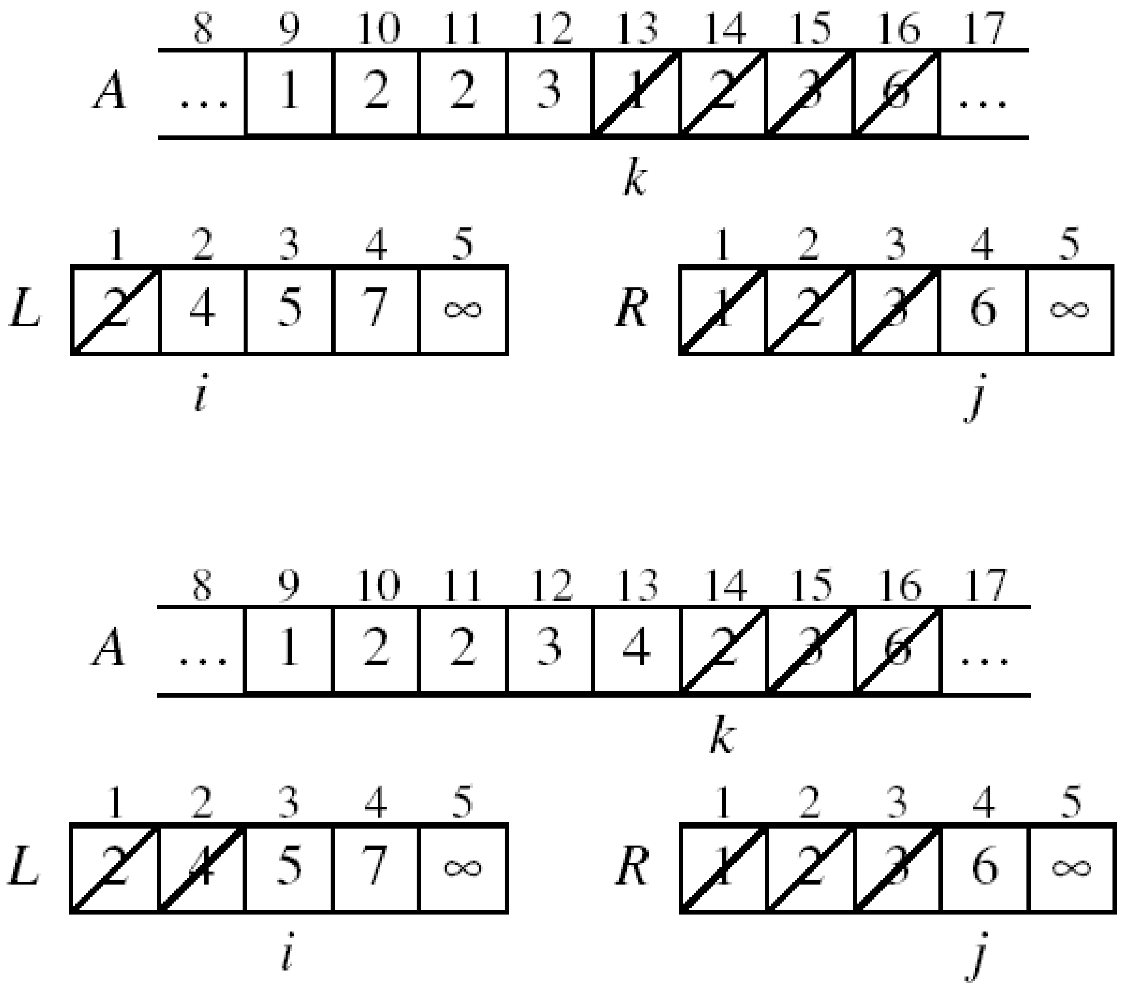


Example: MERGE(A, 9, 12, 16)





Example (cont.)



Example (cont.)

	8	9	10	11	12	13	14	15	16	17
A	…	1	2	2	3	4	5	~~3~~	~~6~~	…
								k		

	1	2	3	4	5
L	~~2~~	~~4~~	~~5~~	7	∞
				i	

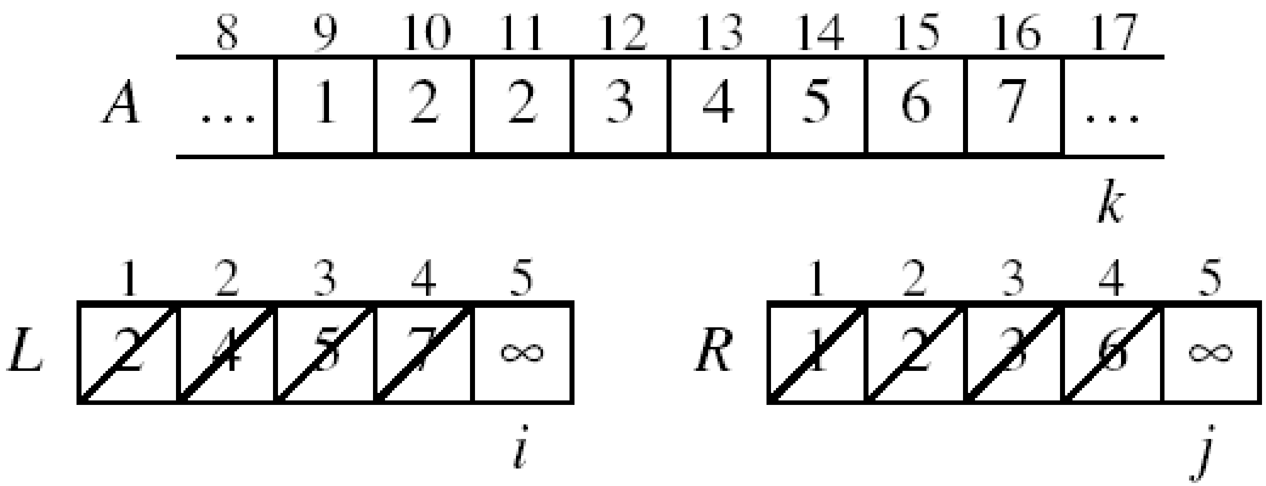
	1	2	3	4	5
R	~~1~~	~~2~~	~~3~~	6	∞
				j	

	8	9	10	11	12	13	14	15	16	17
A	…	1	2	2	3	4	5	6	~~6~~	…
									k	

	1	2	3	4	5
L	~~2~~	~~4~~	~~5~~	7	∞
				i	

	1	2	3	4	5
R	~~1~~	~~2~~	~~3~~	~~6~~	∞
					j

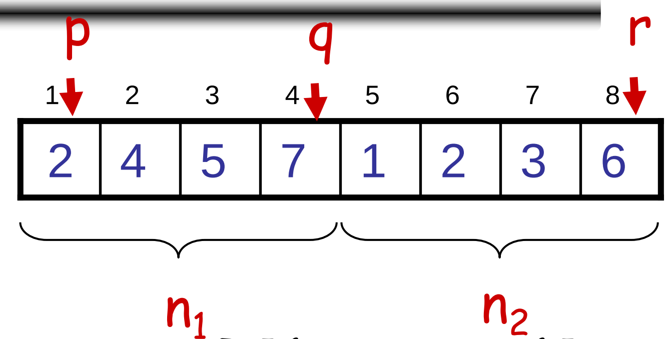
Example (cont.)



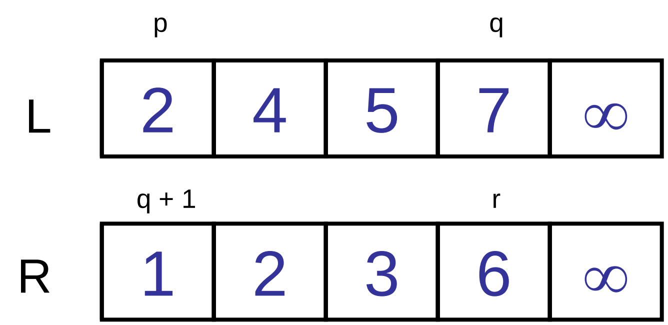
Done!

Merge - Pseudocode

Alg.: MERGE(A, p, q, r)

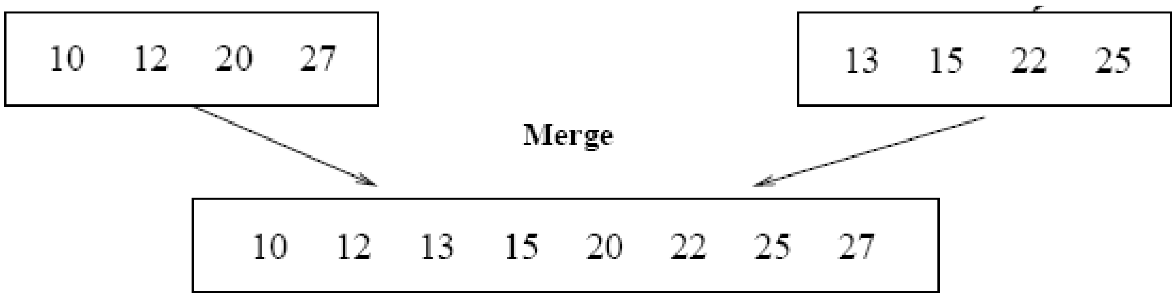


1. Compute n_1 and n_2
2. Copy the first n_1 elements into $L[1\ ..\ n_1 + 1]$ and the next n_2 elements into $R[1\ ..\ n_2 + 1]$
3. $L[n_1 + 1] \leftarrow \infty$; $R[n_2 + 1] \leftarrow \infty$
4. $i \leftarrow 1$; $j \leftarrow 1$
5. **for** $k \leftarrow p$ **to** r
6. **do if** $L[\ i\ ] \leq R[\ j\ ]$
7. **then** $A[k] \leftarrow L[\ i\ ]$
8. $i \leftarrow i + 1$
9. **else** $A[k] \leftarrow R[\ j\ ]$
10. $j \leftarrow j + 1$



Running Time of Merge (assume last **for** loop)

- Initialization (copying into temporary arrays):
  - $\Theta(n_1 + n_2) = \Theta(n)$
- Adding the elements to the final array:
  - n iterations, each taking constant time $\Rightarrow \Theta(n)$
- Total time for Merge:
  - $\Theta(n)$



Analyzing Divide-and Conquer Algorithms

- The recurrence is based on the three steps of the paradigm:
  - $T(n)$ – running time on a problem of size n
  - **Divide** the problem into a subproblems, each of size **n/b**: takes $D(n)$
  - **Conquer** (solve) the subproblems $aT(n/b)$
  - **Combine** the solutions $C(n)$

$$T(n) = \begin{cases} \Theta(1) & \text{if } n \le c \\ aT(n/b) + D(n) + C(n) & \text{otherwise} \end{cases}$$

MERGE-SORT Running Time

- **Divide:**
  - compute q as the average of p and r: $D(n) = \Theta(1)$
- **Conquer:**
  - recursively solve 2 subproblems, each of size $n/2$
    $\Rightarrow 2T(n/2)$
- **Combine:**
  - MERGE on an n-element subarray takes $\Theta(n)$ time
    $\Rightarrow C(n) = \Theta(n)$

$$T(n) = \begin{cases} \Theta(1) & \text{if } n = 1 \\ 2T(n/2) + \Theta(n) & \text{if } n > 1 \end{cases}$$

Solve the Recurrence

$$T(n) = \begin{cases} c & \text{if } n = 1 \\ 2T(n/2) + cn & \text{if } n > 1 \end{cases}$$

Use Master's Theorem:

Compare n with $f(n) = cn$

Case 2: $T(n) = \Theta(n \lg n)$

Merge Sort - Discussion

- Running time insensitive of the input

- Advantages:
  – Guaranteed to run in $\Theta(nlgn)$

- Disadvantage
  – Requires extra space $\approx N$

Sorting Challenge 1

Problem: Sort a file of huge records with tiny keys

Example application: Reorganize your MP-3 files

Which method to use?

A. merge sort, guaranteed to run in time ~NlgN
B. selection sort
C. bubble sort
D. a custom algorithm for huge records/tiny keys
E. insertion sort

Sorting Files with Huge Records and Small Keys

- Insertion sort or bubble sort?
  - NO, too many exchanges
- Selection sort?
  - YES, it takes **linear** time for exchanges
- Merge sort or custom method?
  - Probably not: selection sort simpler, does less swaps

Sorting Challenge 2

Problem: Sort a huge randomly-ordered file of small records

Application: Process transaction record for a phone company

Which sorting method to use?

A. Bubble sort
B. Selection sort
C. Mergesort guaranteed to run in time ~N*lg*N
D. Insertion sort

Sorting Huge, Randomly - Ordered Files

- Selection sort?
  - NO, always takes quadratic time
- Bubble sort?
  - NO, quadratic time for randomly-ordered keys
- Insertion sort?
  - NO, quadratic time for randomly-ordered keys
- Merge sort?
  - YES, it is designed for this problem

Sorting Challenge 3

Problem: sort a file that is already almost in order

Applications:

– Re-sort a huge database after a few changes
– Doublecheck that someone else sorted a file

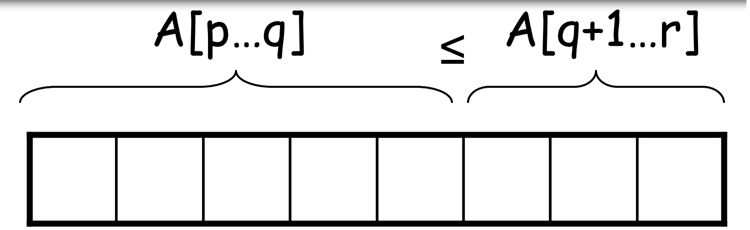
Which sorting method to use?

A. Mergesort, guaranteed to run in time ~NlgN
B. Selection sort
C. Bubble sort
D. A custom algorithm for almost in-order files
E. Insertion sort

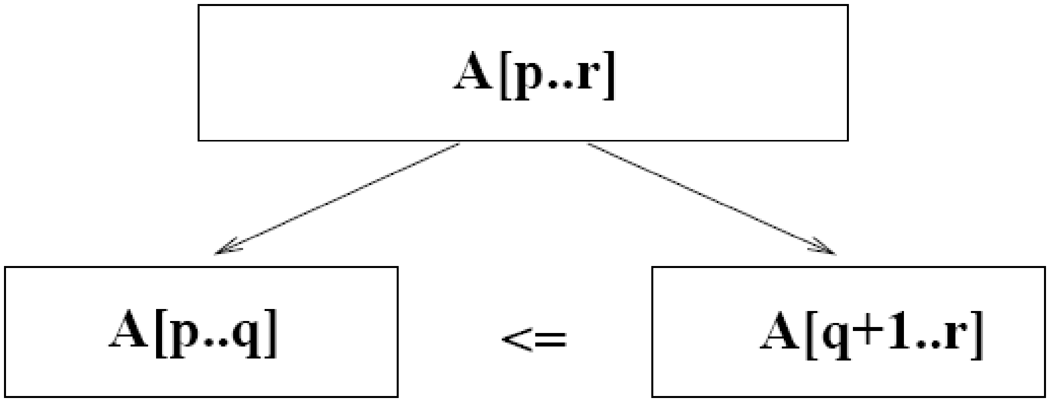
Sorting Files That are Almost in Order

- Selection sort?
  – NO, always takes quadratic time
- Bubble sort?
  – NO, bad for some definitions of “almost in order”
  – Ex: B C D E F G H I J K L M N O P Q R S T U V W X Y Z A
- Insertion sort?
  – **YES, takes linear time for most definitions of “almost in order”**
- Mergesort or custom method?
  – Probably not: insertion sort simpler and faster

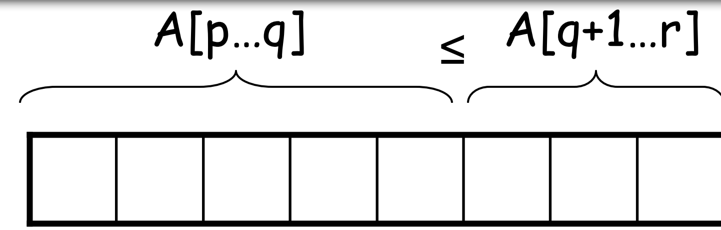
Quicksort



- Sort an array $A[p...r]$
- **Divide**
  - Partition the array A into 2 subarrays $A[p..q]$ and $A[q+1..r]$, such that each element of $A[p..q]$ is **smaller than or equal** to each element in $A[q+1..r]$.
  - Need to find index q to partition the array



Quicksort



- **Conquer**
  - Recursively sort $A[p..q]$ and $A[q+1..r]$ using Quicksort
- **Combine**
  - Trivial: the arrays are sorted in place
  - No additional work is required to combine them
  - The entire array is now sorted

QUICKSORT

Alg.: QUICKSORT(A, p, r)

Initially: p=1, r=n

if p < r

then $q \leftarrow$ PARTITION(A, p, r)

QUICKSORT (A, p, q)

QUICKSORT (A, q+1, r)

Recurrence:

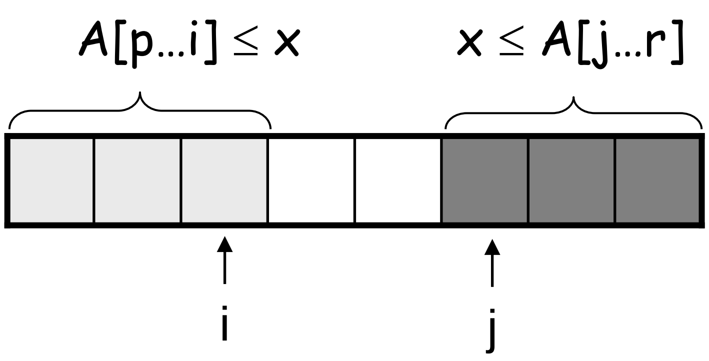
T(n) = T(q) + T(n – q) + f(n) $(f(n)$ depends on PARTITION())

Partitioning the Array

- Choosing PARTITION()
  - There are different ways to do this
  - Each has its own advantages/disadvantages
- Hoare partition
- Select a pivot element **x** around which to partition
  - Grows two regions

    $A[p \ldots i] \le \mathbf{x}$

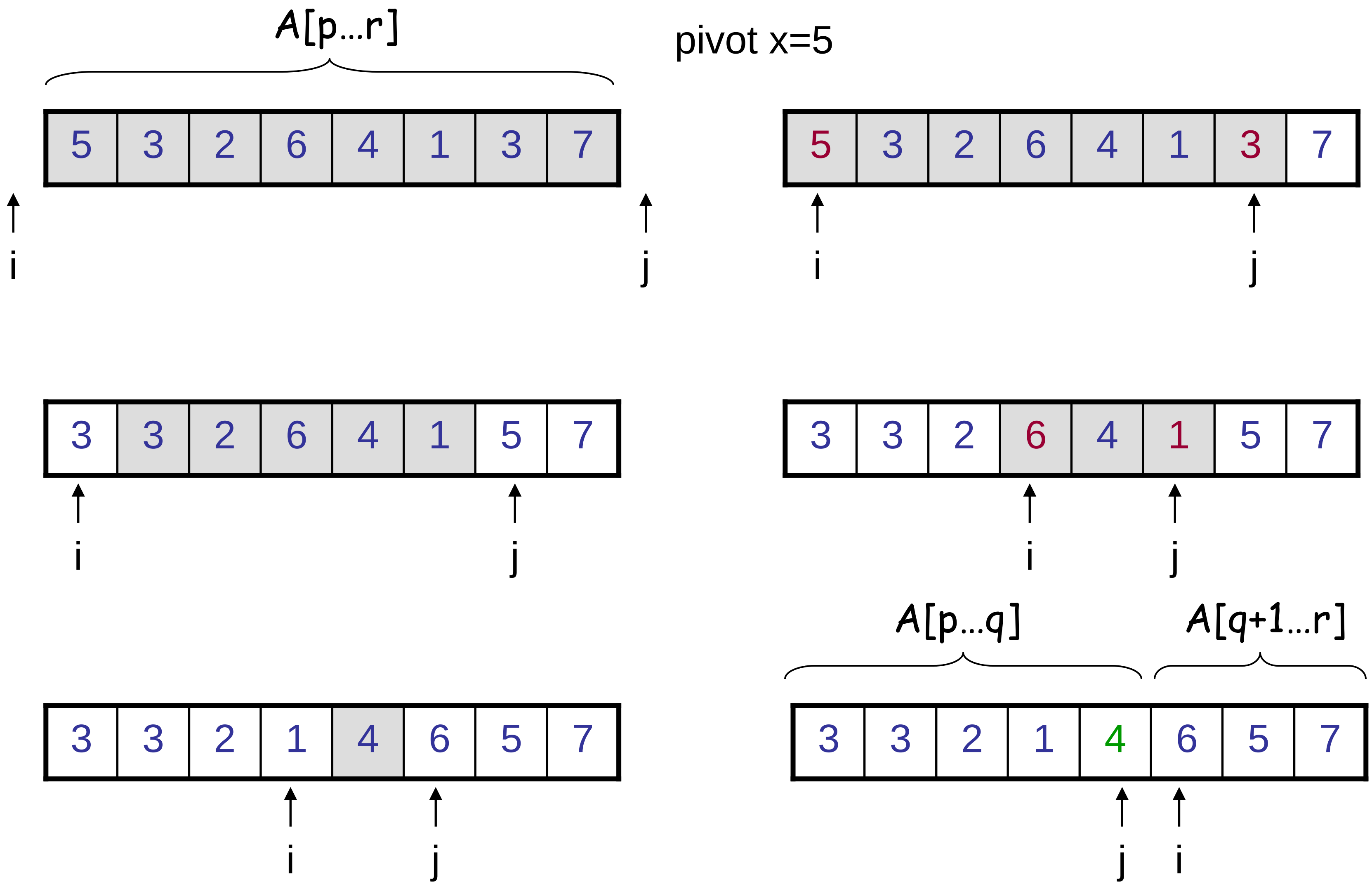
    $\mathbf{x} \le A[j \ldots r]$



Hoare partition

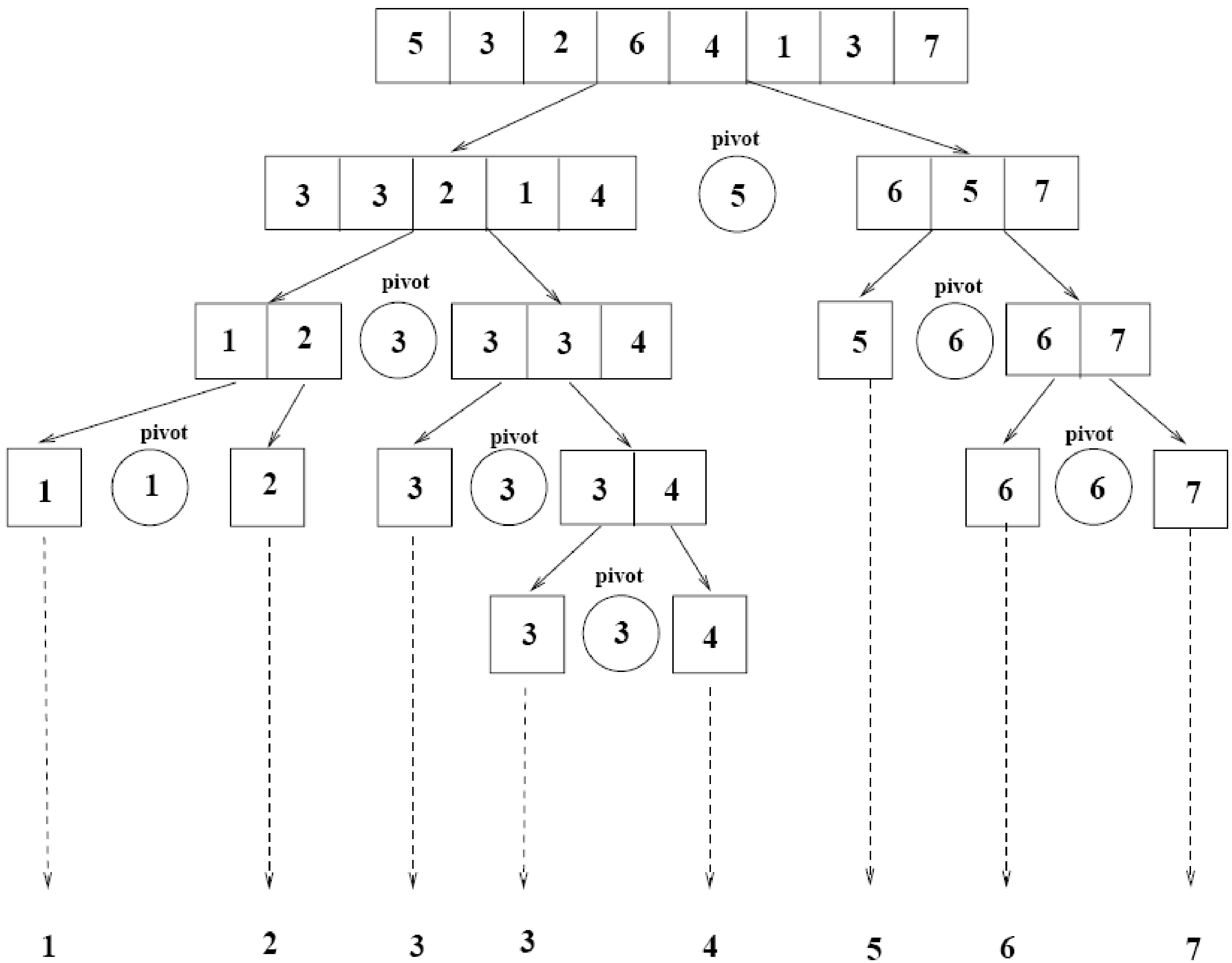
- Hoare Partition uses a Two-directional scanning technique that comes from the left until it finds an element that is bigger than the pivot, and from the right until it finds an element that is smaller than the pivot and then swaps the two.
- The process continues until the scan from the left meets the scan from the right.

Example



A[p...r]
pivot x=5
5 3 2 6 4 1 3 7
i
j
5 3 2 6 4 1 3 7
i
j
3 3 2 6 4 1 5 7
i
j
3 3 2 6 4 1 5 7
i
j
3 3 2 1 4 6 5 7
i
j
A[p...q]
A[q+1...r]
3 3 2 1 4 6 5 7
j
i

Example

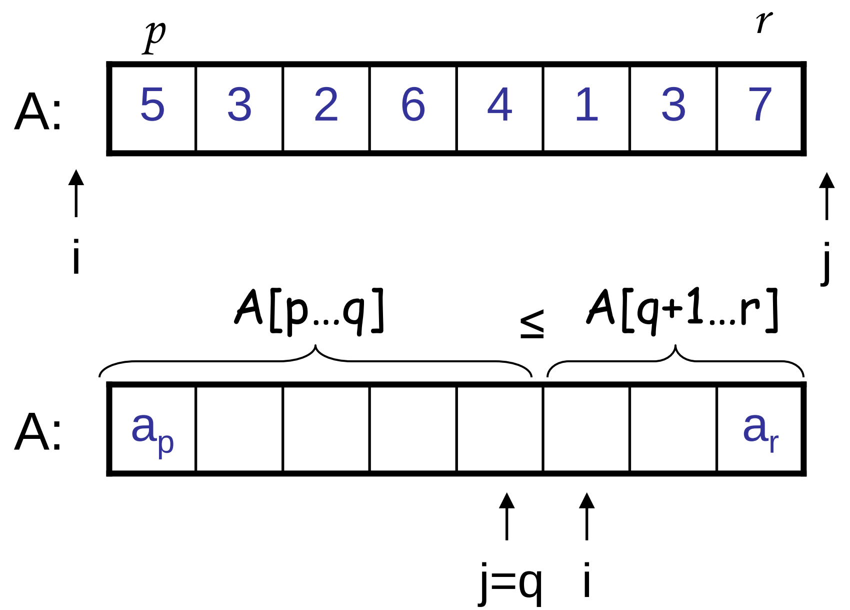


5 3 2 6 4 1 3 7
pivot
3 3 2 1 4
5
6 5 7
pivot
1 2
3
3 3 4
pivot
5
6
6 7
pivot
1
1
2
pivot
3
3
3 4
pivot
6
6
7
pivot
3
3
4
1 2 3 3 4 5 6 7

Partitioning the Array

Alg. PARTITION (A, p, r)

```
x ← A[p]
i ← p - 1
j ← r + 1
while TRUE
        do repeat j ← j - 1
                until A[j] ≤ x
        do  repeat i ← i + 1
                until A[i] ≥ x
         if i < j
                then exchange A[i] ↔ A[j]
         else return j
```



Each element is visited once!

Running time: $\Theta(n)$

$n = r - p + 1$

Recurrence

Alg.: QUICKSORT(A, p, r)   Initially: p=1, r=n

if p < r

then $q \leftarrow$ PARTITION(A, p, r)

QUICKSORT (A, p, q)

QUICKSORT (A, q+1, r)

Recurrence:

$$T(n) = T(q) + T(n - q) + n$$

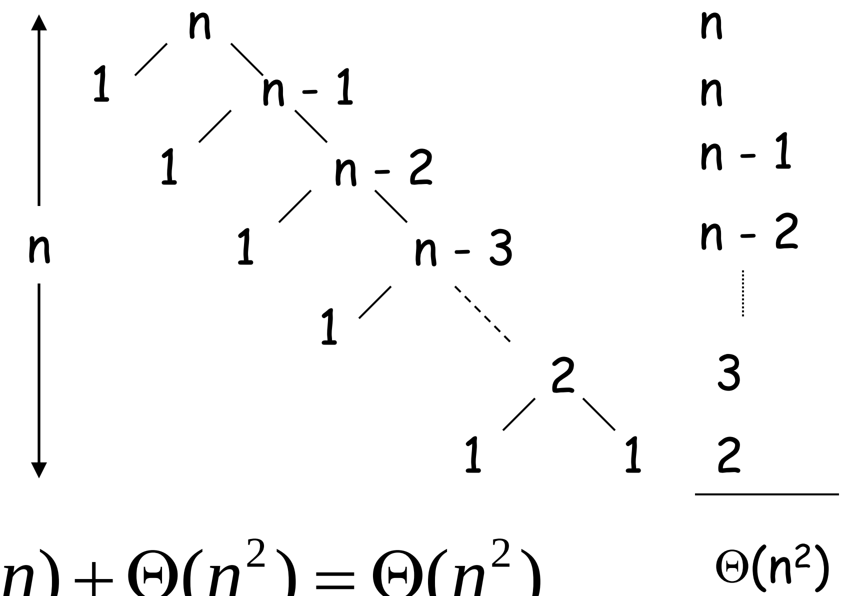
Worst Case Partitioning

- Worst-case partitioning
  - One region has one element and the other has n – 1 elements
  - Maximally unbalanced
- Recurrence: q=1

  T(n) = T(1) + T(n – 1) + n,

  T(1) = Θ(1)

  T(n) = T(n – 1) + n



$$= \quad n + \left(\sum_{k=1}^{n} k \right) - 1 = \Theta(n) + \Theta(n^2) = \Theta(n^2)$$

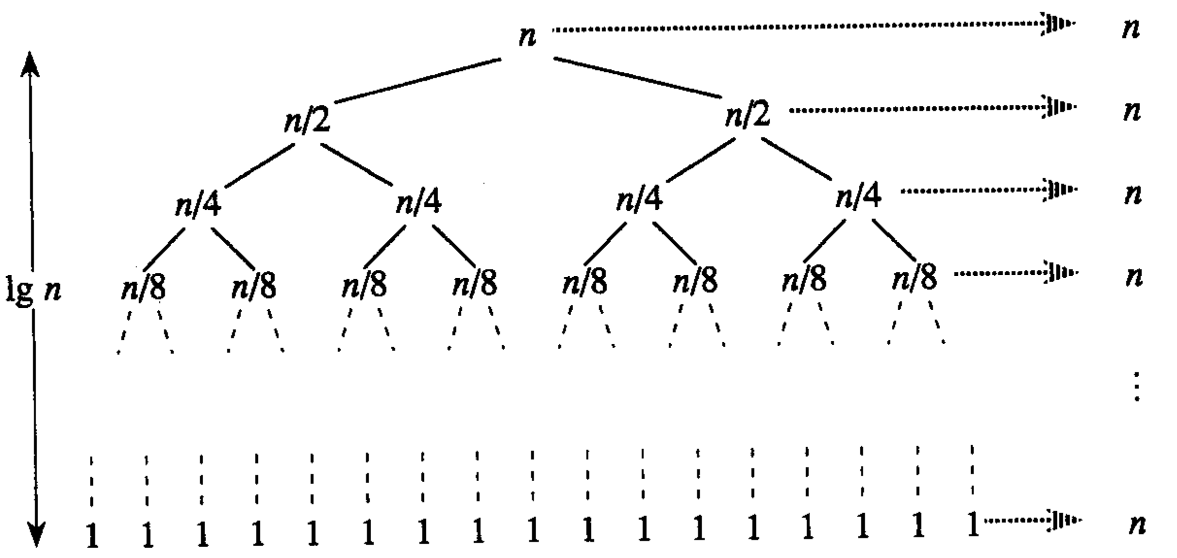
When does the worst case happen?

Best Case Partitioning

- Best-case partitioning
  - Partitioning produces two regions of size $n/2$
- Recurrence: q=n/2

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

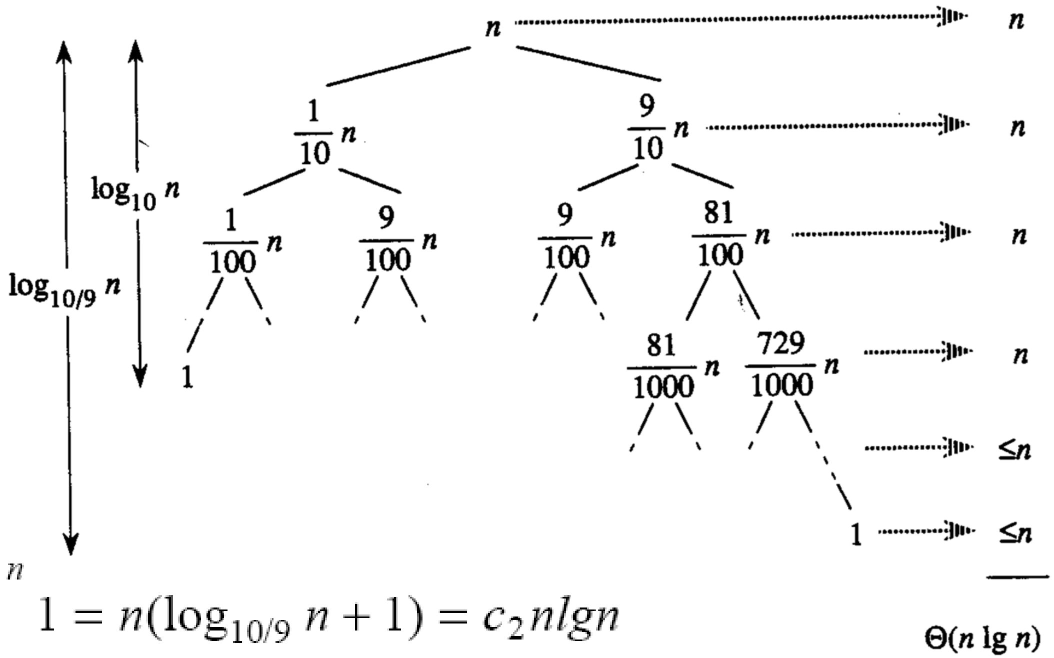
$T(n) = \Theta(n\lg n)$ (Master theorem)



Case Between Worst and Best

- 9-to-1 proportional split

Q(n) = Q(9n/10) + Q(n/10) + n



- Using the recursion tree:

longest path: $Q(n) \le n \sum_{i=0}^{\log_{10/9} n} 1 = n(\log_{10/9} n + 1) = c_2 n lgn$

shortest path: $Q(n) \ge n \sum_{i=0}^{\log_{10} n} 1 = n \log_{10} n = c_1 n lgn$

Thus, $Q(n) = \Theta(nlgn)$

How does partition affect performance?

- **Any splitting of constant proportionality** yields $\Theta(n\lg n)$ time !!!

- Consider the $(1 : n-1)$ splitting:

$$\text{ratio}=1/(n-1) \text{ not a constant !!!}$$

- Consider the $(n/2 : n/2)$ splitting:

$$\text{ratio}=(n/2)/(n/2) = 1 \text{ it is a constant !!}$$

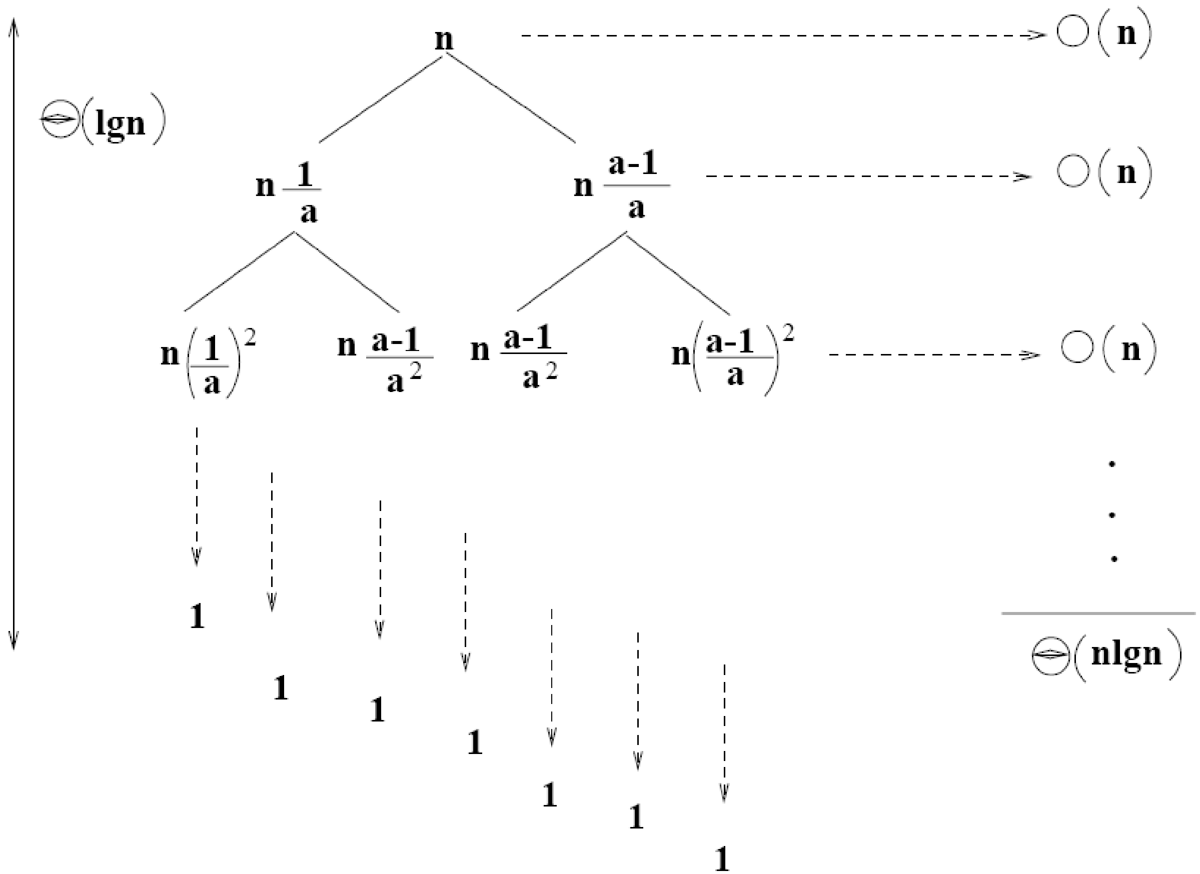
- Consider the $(9n/10 : n/10)$ splitting:

$$\text{ratio}=(9n/10)/(n/10) = 9 \text{ it is a constant !!}$$

How does partition affect performance?

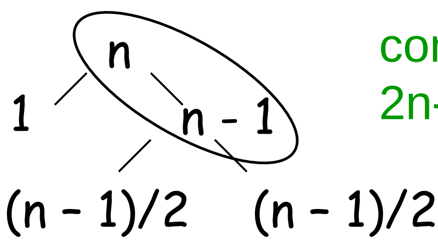
- Any $((a-1)n/a : n/a)$ splitting:

$$\text{ratio}=((a-1)n/a)/(n/a) = a-1 \text{ it is a constant !!}$$



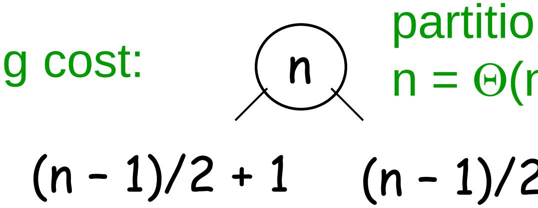
Performance of Quicksort

- Average case
  - All permutations of the input numbers are equally likely
  - On a random input array, we will have a **mix** of well balanced and unbalanced splits
  - Good and bad splits are randomly distributed across throughout the tree



combined partitioning cost:
$2n-1 = \Theta(n)$

Alternate of a good and a bad split



partitioning cost:
$n = \Theta(n)$

Nearly well balanced split

- Running time of Quicksort when levels alternate between good and bad splits is $O(n\lg n)$

Randomizing QuickSort

(Appendix C.2 , Appendix C.3)

(Chapter 5, Chapter 7)

Randomizing Quicksort

- Randomly permute the elements of the input array before sorting
- OR ... modify the PARTITION procedure
  - At each step of the algorithm we exchange element $A[p]$ with an element chosen at random from $A[p...r]$.
  - The pivot element $x = A[p]$ is equally **likely to be any** one of the $r - p + 1$ elements of the subarray.

Randomized Algorithms

- No input can elicit worst case behavior
  - Worst case occurs only if we get “unlucky” numbers from the random number generator
- Worst case becomes less likely
  - Randomization can <u>NOT</u> eliminate the worst-case but it can make it less likely!

Randomized PARTITION

Alg.: RANDOMIZED-PARTITION(A, p, r)

i ← RANDOM(p, r)

exchange A[p] $\boxed{\leftrightarrow}$ A[i]

return PARTITION(A, p, r)

Randomized Quicksort

Alg. : RANDOMIZED-QUICKSORT(A, p, r)

if $p < r$

then $q \leftarrow$ RANDOMIZED-PARTITION(A, p, r)

RANDOMIZED-QUICKSORT(A, p, q)

RANDOMIZED-QUICKSORT(A, $q + 1$, r)

Formal Worst-Case Analysis of Quicksort

- $T(n)$ = worst-case running time

$$T(n) = \max_{1 \le q \le n-1} (T(q) + T(n-q)) + \Theta(n)$$

- Use the substitution method to show that the running time of Quicksort is $O(n^2)$
- Guess $T(n) = O(n^2)$
  - Induction goal: $T(n) \le cn^2$
  - Induction hypothesis: $T(k) \le ck^2$ for any $k < n$

Worst-Case Analysis of Quicksort

- Proof of induction goal:

$$T(n) \le \max_{1 \le q \le n-1} (cq^2 + c(n-q)^2) + \Theta(n) =$$

$$= c \cdot \max_{1 \le q \le n-1} (q^2 + (n-q)^2) + \Theta(n)$$

- The expression $q^2 + (n-q)^2$ achieves a maximum over the range $1 \le q \le n-1$ at one of the endpoints

$$\max_{1 \le q \le n-1} (q^2 + (n - q)^2) = 1^2 + (n - 1)^2 = n^2 - 2(n - 1)$$

$$T(n) \le cn^2 - 2c(n - 1) + \Theta(n)$$

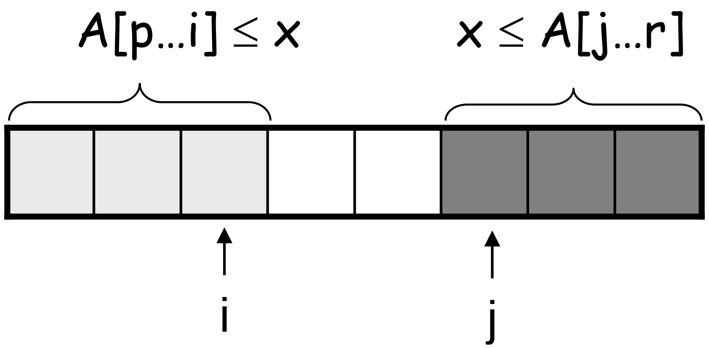
$$\le cn^2$$

Revisit Partitioning

- Hoare's partition
  - Select a pivot element **x** around which to partition
  - Grows two regions

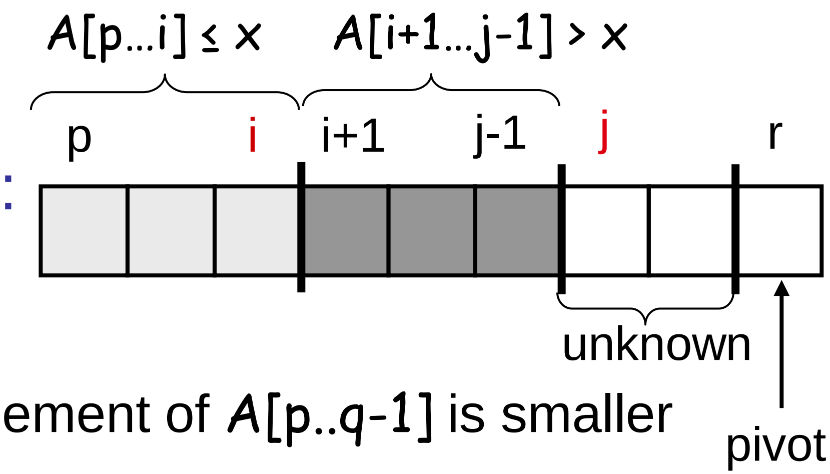
    $A[p \ldots i] \le x$

    $x \le A[j \ldots r]$

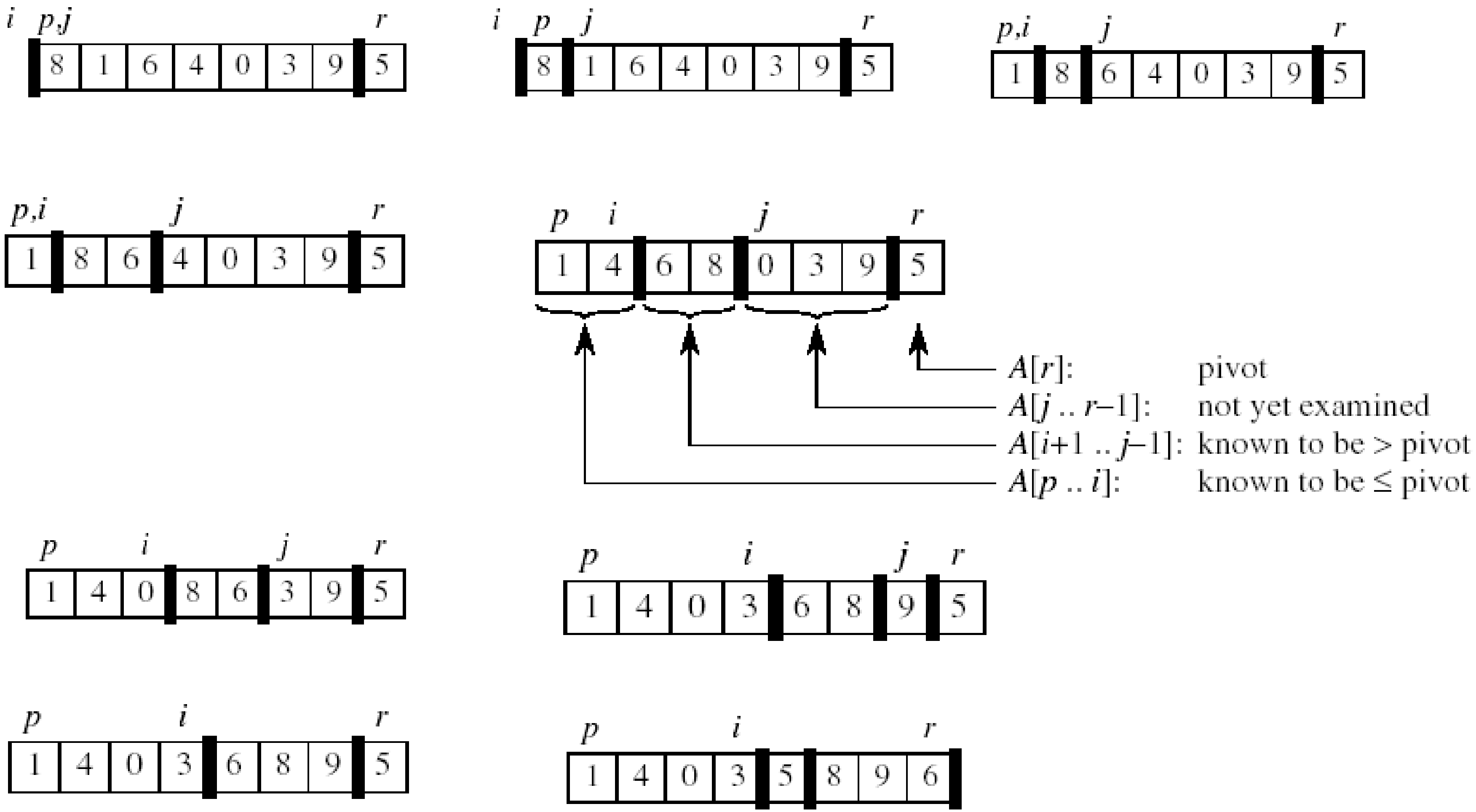


Another Way to PARTITION (Lomuto's partition – page 146)

- Given an array A, partition the array into the following subarrays:
  - A pivot element $x = A[q]$
  - Subarray $A[p..q-1]$ such that each element of $A[p..q-1]$ is smaller than or equal to x (the pivot)
  - Subarray $A[q+1..r]$, such that each element of $A[p..q+1]$ is **strictly** greater than x (the pivot)
- The pivot element is not included in any of the two subarrays



Example



at the end, swap pivot

Another Way to PARTITION (cont'd)

Alg.: PARTITION*(A, p, r)*

$x \leftarrow A[r]$

$i \leftarrow p - 1$

for $j \leftarrow p$ **to** $r - 1$ **do**

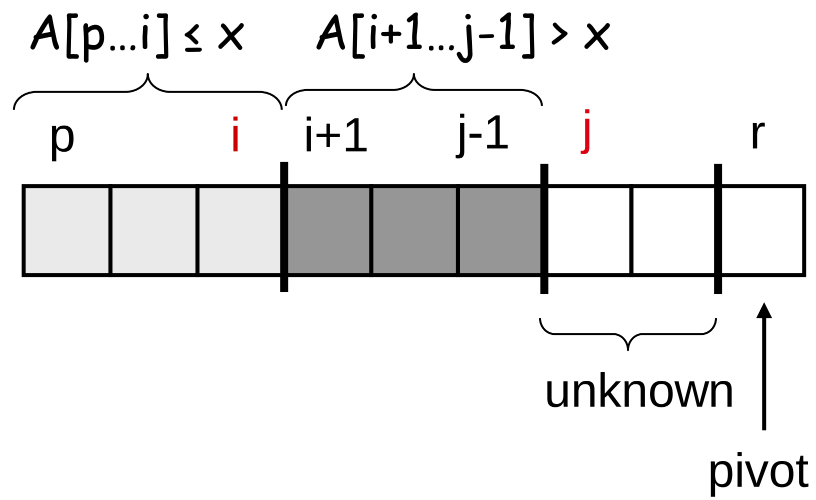
  if $A[j] \le x$ **then**

    $i \leftarrow i + 1$

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

exchange $A[i + 1] \leftrightarrow A[r]$

return $i + 1$



Chooses the last element of the array as a pivot
Grows a subarray [p..i] of elements ≤ x
Grows a subarray [i+1..j-1] of elements >x
Running Time: $\Theta(n)$, where n=r-p+1

Randomized Quicksort (using Lomuto's partition)

Alg. : RANDOMIZED-QUICKSORT(A, p, r)

if p < r

then $q \leftarrow$ RANDOMIZED-PARTITION(A, p, r)

RANDOMIZED-QUICKSORT(A, p, $q - 1$)

RANDOMIZED-QUICKSORT(A, $q + 1$, r)

The pivot is <u>no longer</u> included in any of the subarrays!!

Analysis of Randomized Quicksort

Alg. : RANDOMIZED-QUICKSORT(A, p, r)

The running time of Quicksort is dominated by PARTITION !!

if p < r

then $q \leftarrow$ RANDOMIZED-PARTITION(A, p, r)

RANDOMIZED-QUICKSORT(A, p, q - 1)

RANDOMIZED-QUICKSORT(A, q + 1, r)

PARTITION is called at most n times

(at each call a pivot is selected and never again included in future calls)

PARTITION

Alg.: PARTITION*(A, p, r)*

$x \leftarrow A[r]$

$i \leftarrow p - 1$

O(1) - constant

for $j \leftarrow p$ **to** $r - 1$

  do if $A[\, j \,] \leq x$

    then $i \leftarrow i + 1$

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

# of comparisons: X_k between the pivot and the other elements

exchange $A[i + 1] \leftrightarrow A[r]$

return $i + 1$

O(1) - constant

Amount of work at call k: $c + X_k$

Average-Case Analysis of Quicksort

- Let X = **total number of comparisons performed in <u>all calls</u> to PARTITION:** $X = \sum_k X_k$

- The total work done over the entire execution of Quicksort is

$$O(nc+X)=O(n+X)$$

- Need to estimate E(X)

Review of Probabilities

- **Definitions**

  - random experiment: an experiment whose result is not certain in advance (e.g., throwing a die)

  - outcome: the result of a random experiment

  - sample space: the set of all possible outcomes (e.g., $\{1,2,3,4,5,6\}$)

  - event: a subset of the sample space (e.g., obtain an odd number in the experiment of throwing a die = $\{1,3,5\}$)

Review of Probabilities

- **Probability of an event** (discrete case)

  - The likelihood that an event will occur if the underlying random experiment is performed

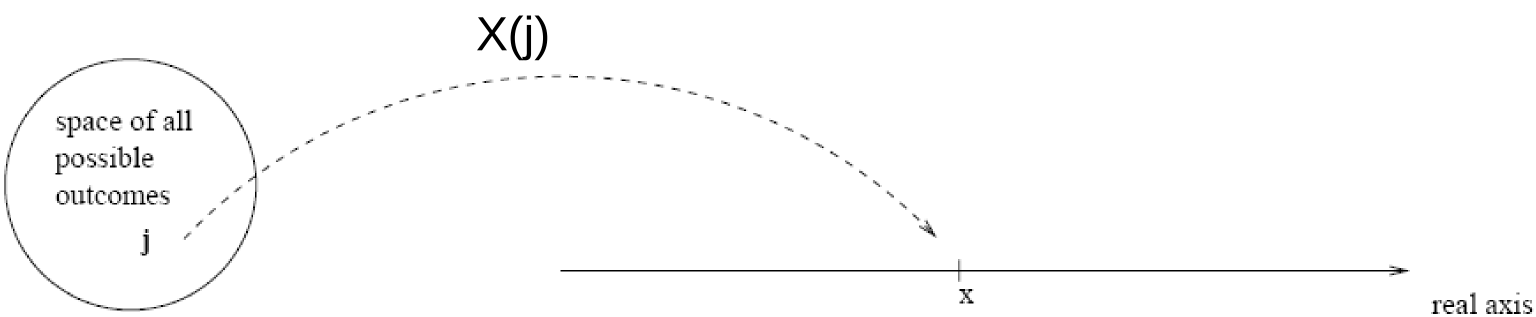
$$P(event) = \frac{number\ of\ favorable\ outcomes}{total\ number\ of\ possible\ outcomes}$$

Example: $P(obtain\ an\ odd\ number) = 3/6 = 1/2$

Random Variables

*Def.: **(Discrete) random variable X**: a function from a sample space S to the real numbers.*

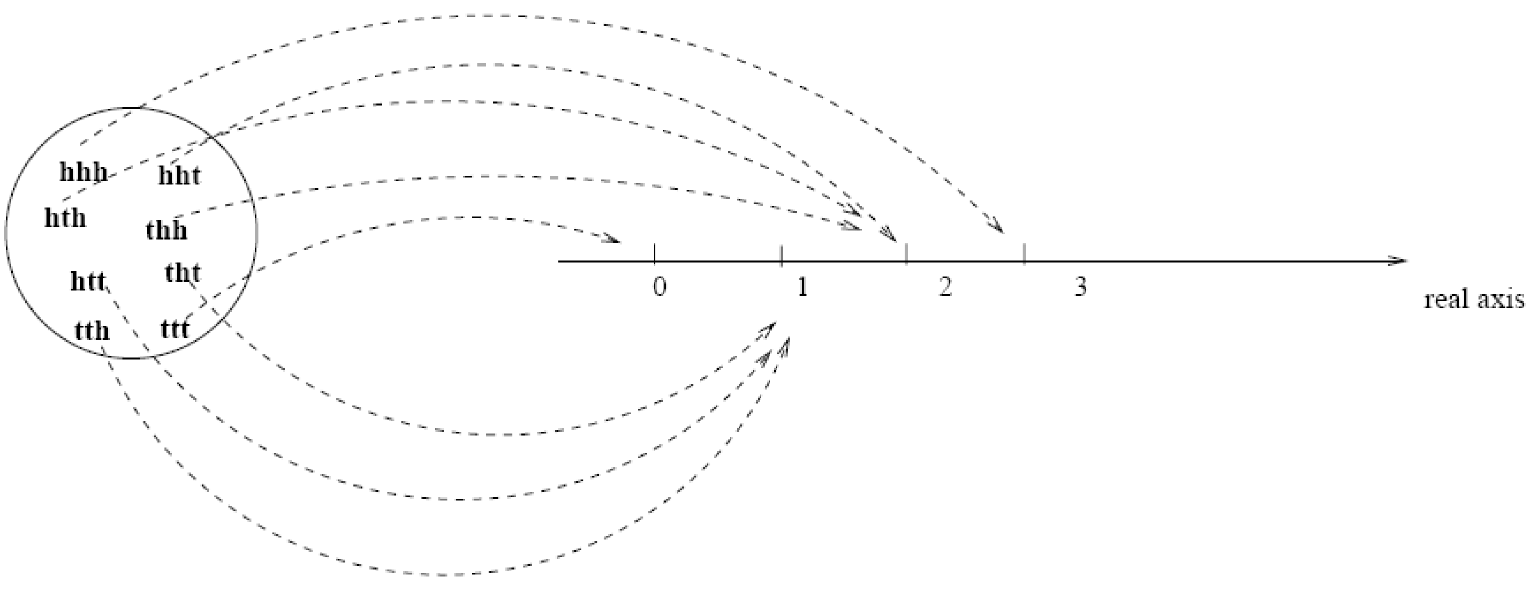
– It associates a real number with each possible outcome of an experiment.



Random Variables

E.g.: Toss a coin three times

define X = “numbers of heads”



Computing Probabilities Using Random Variables

- Example: consider the experiment of throwing a pair of dice

Define the r.v. X="sum of dice"

$X = x$ corresponds to the event $A_x = \{s \in S / X(s) = x\}$

(e.g., $X = 5$ corresponds to $A_5 = \{(1,4),(4,1),(2,3),(3,2)\}$

$$P(X = x) = P(A_x) = \sum_{s:X(s)=x} P(s)$$

$$(P(X = 5) = P((1,4)) + P((4,1)) + P((2,3)) + P((2,3)) = 4/36 = 1/9)$$

Expectation

- Expected value (expectation, mean) of a discrete random variable X is:

$$E[X] = \Sigma_x \, x \Pr\{X = x\}$$

  – "Average" over all possible values of random variable X

Examples

Example: X = face of one fair dice

E[X] = 1·1/6 + 2·1/6 + 3·1/6 + 4·1/6 + 5·1/6 +
6·1/6 = 3.5

Example: X="sum of dice"

Events												
Sum	1	2	3	4	5	6	7	8	9	10	11	12
Probability	0/36	1/36	2/36	3/36	4/36	5/35	6/36	5/36	4/360	3/36	2/36	1/36

$$E(X) = 1P(X = 1) + 2P(X = 2) + \ldots + 12P(X = 12) = (0 + 2 + \ldots + 12)/36 = 7$$

Indicator Random Variables

- Given a sample space S and an event *A*, we define the ***indicator random variable*** I{A} associated with A:
  - $I\{A\} = \begin{cases} 1 & \text{if A occurs} \\ 0 & \text{if A does not occur} \end{cases}$
- The expected value of an indicator random variable $X_A = I\{A\}$ is:

$$E[X_A] = Pr\,\{A\}$$

- Proof:

$$E[X_A] = E[I\{A\}] = 1 * Pr\{A\} + 0 * Pr\{\bar{A}\} = Pr\{A\}$$

Average-Case Analysis of Quicksort

- Let X = **total number of comparisons performed in all calls to PARTITION:** $X = \sum_k X_k$

- The total work done over the **entire** execution of Quicksort is

  O(n+X)

- Need to estimate E(X)

Notation

z_2	z_9	z_8	z_3	z_5	z_4	z_1	z_6	z_{10}	z_7
2	9	8	3	5	4	1	6	10	7

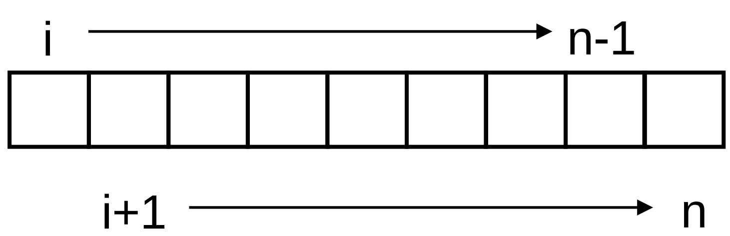
- Rename the elements of A as $z_1, z_2, \ldots, z_n$, with z_i being the <u>*i*-th smallest</u> element
- Define the set $Z_{ij} = \{z_i, z_{i+1}, \ldots, z_j\}$ the set of elements between z_i and z_j, inclusive

Total Number of Comparisons in PARTITION

- Define $X_{ij} = I\{z_i \text{ is compared to } z_j\}$

- Total number of comparisons X performed by the algorithm:

$$X = \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} X_{ij}$$



Expected Number of Total Comparisons in PARTITION

- Compute the **expected value of X:**

$$E[X] = E\left[\sum_{i=1}^{n-1}\sum_{j=i+1}^{n} X_{ij}\right] = \sum_{i=1}^{n-1}\sum_{j=i+1}^{n} E\left[X_{ij}\right] =$$

by linearity
of expectation

indicator
random variable

$$= \sum_{i=1}^{n-1}\sum_{j=i+1}^{n} \Pr\{z_i \text{ is compared to } z_j\}$$

the expectation of X_{ij} is equal
to the probability of the event
"z_i is compared to z_j"

Comparisons in PARTITION: Observation 1

- Each pair of elements is compared **at most once** during the entire execution of the algorithm
  - Elements are compared only to the pivot point!
  - Pivot point is excluded from future calls to PARTITION

Comparisons in PARTITION: Observation 2

- Only the pivot is compared with elements in both partitions!

z_2	z_9	z_8	z_3	z_5	z_4	z_1	z_6	z_{10}	z_7
2	9	8	3	5	4	1	6	10	7

$Z_{1,6} = \{1, 2, 3, 4, 5, 6\}$ $\{7\}$ $Z_{8,9} = \{8, 9, 10\}$

pivot

Elements between different partitions are <u>never</u> compared!

Comparisons in PARTITION

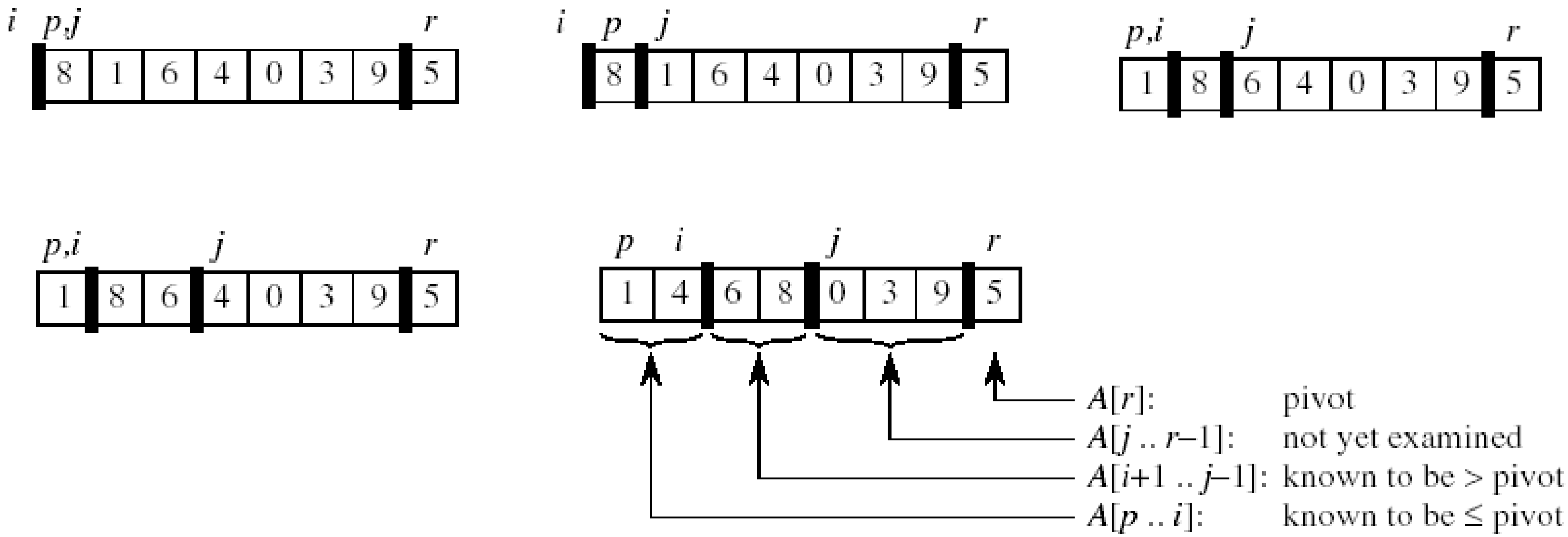
z_2	z_9	z_8	z_3	z_5	z_4	z_1	z_6	z_{10}	z_7
2	9	8	3	5	4	1	6	10	7

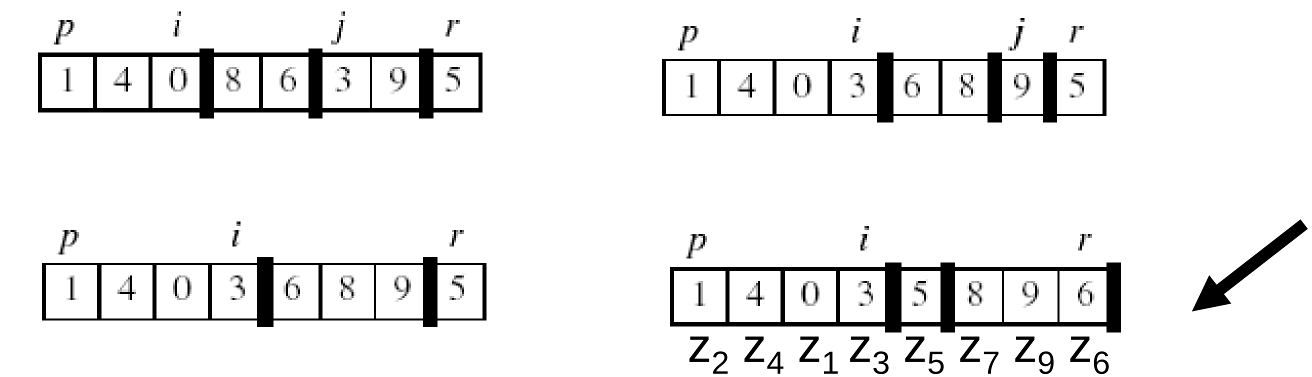
$Z_{1,6}$= {1, 2, 3, 4, 5, 6}   {7} $Z_{8,9}$ = {8, 9, 10}

$$\Pr\{z_i \text{ is compared to } z_j\}?$$

- Case 1: pivot chosen such as: $z_i < x < z_j$
  - z_i and z_j will never be compared
- Case 2: z_i or z_j is the pivot
  - z_i and z_j will be compared
  - only if one of them is chosen as pivot before any other element in range z_i to z_j

See why ☺





z2 will never be compared with z6 since z5 (which belongs to [z_2, z_6]) was chosen as a pivot first !

Probability of comparing z_i with z_j

Pr{ z_i is compared to z_j } =

Pr{ z_i is the first pivot chosen from Z_{ij} }

+

Pr{ z_j is the first pivot chosen from Z_{ij} }

= 1/(j - i + 1) + 1/(j - i + 1) = 2/(j - i + 1)

- There are $j - i + 1$ elements between z_i and z_j
  - Pivot is chosen randomly and independently
  - The probability that any particular element is the first one chosen is $1/(j - i + 1)$

Number of Comparisons in PARTITION

Expected number of comparisons in PARTITION:

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

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

(set k=j-i) (harmonic series)

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

$\Rightarrow$ Expected running time of Quicksort using RANDOMIZED-PARTITION is $O(n\lg n)$

Problem 1

- What is the running time of Quicksort when all the elements are the same?

Problem 1

- What is the running time of Quicksort when all the elements are the same?

  - Using Hoare partition → best case
    - Split in half every time
    - $T(n)=2T(n/2)+n \rightarrow T(n)=\Theta(n\lg n)$

  - Using Lomuto's partition → worst case
    - 1:n-1 splits every time
    - $T(n)=\Theta(n^2)$

Problem 2

- Consider the problem of determining whether an arbitrary sequence $\{x_1, x_2, ..., x_n\}$ of n numbers contains repeated occurrences of some number. Show that this can be done in **Θ(nlgn)** time.

Problem 2

- Consider the problem of determining whether an arbitrary sequence $\{x_1, x_2, ..., x_n\}$ of *n* numbers contains repeated occurrences of some number. Show that this can be done in **Θ(nlgn)** time.
  - Sort the numbers
    - Θ(nlgn)
  - Scan the sorted sequence from left to right, checking whether two successive elements are the same
    - Θ(n)
  - Total
    - Θ(nlgn)+Θ(n)=Θ(nlgn)

Problem 3

- Can we use Binary Search to improve InsertionSort (i.e., find the correct location to insert A[j]?)

Alg.: INSERTION-SORT(*A*)

for j ← 2 **to** n

  do key ← *A*[j]

    Insert *A*[j] into the sorted sequence *A*[1 . . j -1]

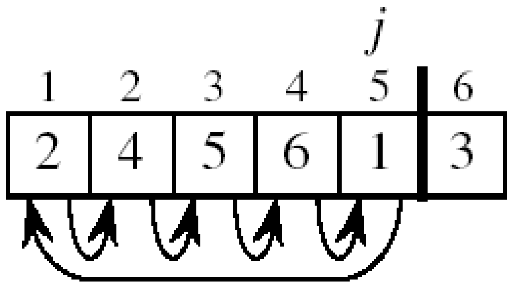
    i ← j - 1

    while i > 0 and *A*[i] > key

      do *A*[i + 1] ← *A*[i]

        i ← i - 1

    A[i + 1] ← key



Problem 3

- Can we use binary search to improve InsertionSort (i.e., find the correct location to insert A[j]?)
  - This idea can reduce the number of comparisons from O(n) to O(lgn)
  - Number of shifts stays the same, i.e., O(n)
  - Overall, time stays the same ...
  - Worthwhile idea when comparisons are expensive (e.g., compare strings)

Problem 4

- Analyze the complexity of the following function:

```
F(i)
 if i=0
   then return 1
 return (2*F(i-1))
```

Problem 4

- Analyze the complexity of the following function:

```
F(i)
 if i=0
   then return 1
 return (2*F(i-1))
```

Recurrence: $T(n)=T(n-1)+c$

Use iteration to solve it .... $T(n)=\Theta(n)$

Lower Bound for Comparison Based Sorting

- Insertion sort, Selection sort, Bubble sort
- Merge sort, Quick sort, Heap sort

Lower Bound for Comparison Based Sorting

Definition

A comparison based sorting algorithm sorts objects by comparing pairs of them.

Lower Bound for Comparison Based Sorting

Definition

A comparison based sorting algorithm sorts objects by comparing pairs of them.

Example

Selection sort and merge sort are comparison based.

Lower Bound for Comparison Based Sorting

Lemma

Any comparison based sorting algorithm performs $\Omega(n \log n)$ comparisons in the worst case to sort n objects.

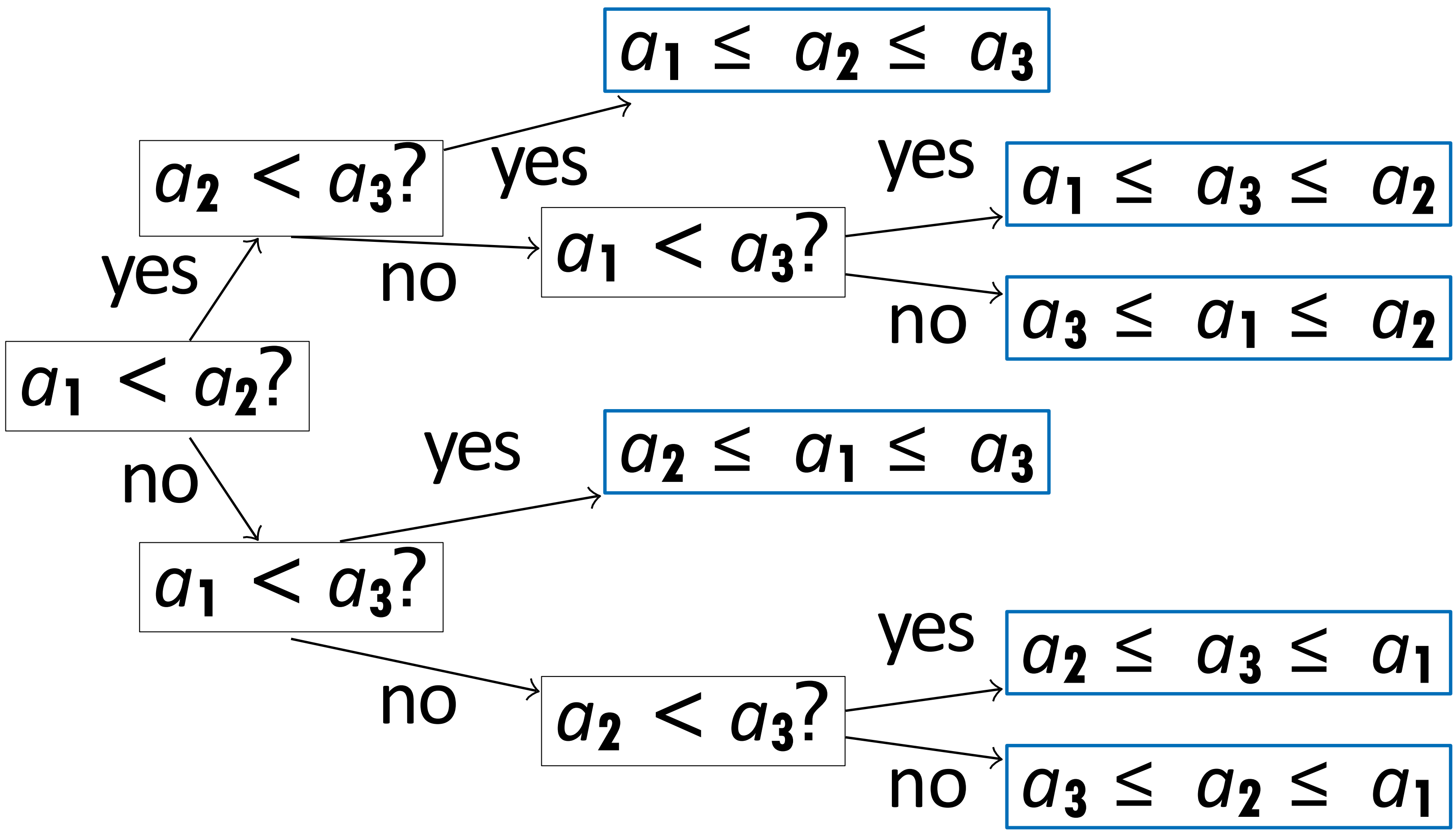
Lemma

Any comparison based sorting algorithm performs $\Omega(n \log n)$ comparisons in the worst case to sort n objects.

In other words

For any comparison based sorting algorithm, there exists an array $A[1 \ldots n]$ such that the algorithm performs at least $\Omega(n \log n)$ comparisons to sort A.

Decision Tree



Estimating Tree Depth

- the number of leaves ℓ in the tree must be at least $n!$ (the total number of permutations)

Estimating Tree Depth

- the number of leaves ℓ in the tree must be at least $n!$ (the total number of permutations)
- the worst-case running time of the algorithm (the number of comparisons made) is at least the depth d

Estimating Tree Depth

- the number of leaves ℓ in the tree must be at least $n!$ (the total number of permutations)
- the worst-case running time of the algorithm (the number of comparisons made) is at least the depth d
- $d \geq \log_2 \ell$ (or, equivalently, $2^d \geq \ell$)

Estimating Tree Depth

- The number of leaves ℓ in the tree must be at least $n!$ (the total number of permutations)
- The worst-case running time of the algorithm (the number of comparisons made) is at least the depth d
- $d \geq \log_2 \ell$ (or, equivalently, $2^d \geq \ell$)
- thus, the running time is at least

$$\log_2(n!) = \Omega(n \log n)$$

Lemma

$$\log_2(n!) = \Omega(n \log n)$$

Proof

$$\begin{aligned}
\log_2(n!) &= \log_2(1 \cdot 2 \cdot \cdots \cdot n) \\
&= \log_2 1 + \log_2 2 + \cdots + \log_2 n \\
&\geq \log_2 \frac{n}{2} + \cdots + \log_2 n \\
&\geq \frac{n}{2} \log_2 \frac{n}{2} = \Omega(n \log n)
\end{aligned}$$

$\square$

Non-Comparison Based Sorting Algorithms

- Counting Sort
- Radix Sort
- Bucket Sort

Comparison Sorting Review

- Insertion sort:
  - Pro's:
    - Easy to code
    - Fast on small inputs (less than ~50 elements)
    - Fast on nearly-sorted inputs
  - Con's:
    - $O(n^2)$ worst case
    - $O(n^2)$ average case

Comparison Sorting Review

- Merge sort:
  - Divide-and-conquer:
    - Split array in half
    - Recursively sort sub-arrays
    - Linear-time merge step
  - Pro's:
    - O(n lg n) worst case
  - Con's:
    - Doesn't sort in place

Comparison Sorting Review

- Heap sort:
  - Uses the very useful heap data structure
    - Complete binary tree
    - Heap property: parent key > children's keys
  - Pro's:
    - O(n lg n) worst case
    - Sorts in place
  - Con's:
    - Fair amount of shuffling memory around

Comparison Sorting Review

- Quick sort:
  - Divide-and-conquer:
    - Partition array into two sub-arrays, recursively sort
    - All of first sub-array < all of second sub-array
  - Pro's:
    - O(n lg n) average case
    - Sorts in place
    - Fast in practice
  - Con's:
    - O(n^2) worst case
      - Naïve implementation: worst case on sorted input
      - Good partitioning makes this very unlikely.

Non-Comparison Based Sorting

- Many times we have <u>restrictions</u> on our <u>keys</u>
  – Social Security Numbers
  – Employee ID's
- We will examine three <u>algorithms</u> which under certain conditions can run in <u>O(*n*)</u> time.
  – Counting sort
  – Radix sort
  – Bucket sort

Example: sorting small integers

	1	2	3	4	5	6	7	8	9	10	11	12
A	2	3	2	1	3	2	2	3	2	2	2	1

Example: sorting small integers

	1	2	3	4	5	6	7	8	9	10	11	12
A	2	3	2	1	3	2	2	3	2	2	2	1

$\downarrow$

	1	2	3
Count	2	7	3

Example: sorting small integers

	1	2	3	4	5	6	7	8	9	10	11	12
A	2	3	2	1	3	2	2	3	2	2	2	1

	1	2	3
Count	2	7	3

	1	2	3	4	5	6	7	8	9	10	11	12
A'	1	1	2	2	2	2	2	2	2	3	3	3

Example: sorting small integers

	1	2	3	4	5	6	7	8	9	10	11	12
A	2	3	2	1	3	2	2	3	2	2	2	1

$\downarrow$

we have sorted these numbers
without actually comparing them!

$\downarrow$

	1	2	3	4	5	6	7	8	9	10	11	12
A'	1	1	2	2	2	2	2	2	2	3	3	3

Counting Sort: Ideas

- Assume that all elements of $A[1 \ldots n]$ are integers from 1 to M.

Counting Sort: Ideas

- Assume that all elements of $A[1 \ldots n]$ are integers from 1 to M.
- By a single scan of the array A, count the number of occurrences of each $1 \leq k \leq M$ in the array A and store it in $Count[k]$.

Counting Sort: Ideas

- Assume that all elements of $A[1 \ldots n]$ are integers from 1 to M.
- By a single scan of the array A, count the number of occurrences of each $1 \leq k \leq M$ in the array A and store it in $Count[k]$.
- Using this information, fill in the sorted array A'.

CountSort($A[1 \ldots n]$)

$Count[1 \ldots M] \leftarrow [0, \ldots, 0]$

for i from 1 to n:

  $Count[A[i]] \leftarrow Count[A[i]] + 1$

{k appears $Count[k]$ times in A}

$Pos[1 \ldots M] \leftarrow [0, \ldots, 0]$

$Pos[1] \leftarrow 1$

for j from 2 to M:

  $Pos[j] \leftarrow Pos[j - 1] + Count[j - 1]$

{k will occupy range $[Pos[k] \ldots Pos[k + 1] - 1]$}

for i from 1 to n:

  $A'[Pos[A[i]]] \leftarrow A[i]$

  $Pos[A[i]] \leftarrow Pos[A[i]] + 1$

Counting Sort

```
CountingSort(A, B, k)
    for i=1 to k
        C[i]= 0;
    for j=1 to n
        C[A[j]] += 1;
    for i=2 to k
        C[i] = C[i] + C[i-1];
    for j=n downto 1
        B[C[A[j]]] = A[j];
        C[A[j]] -= 1;
```

This is called a ***histogram***.

Counting Sort

```
CountingSort(A, B, k)
    for i=1 to k
        C[i]= 0;
    for j=1 to n
        C[A[j]] += 1;
    for i=2 to k
        C[i] = C[i] + C[i-1];
    for j=n downto 1
        B[C[A[j]]] = A[j];
        C[A[j]] -= 1;
```

Takes time O(k)

Takes time O(n)

What is the running time? Total time: O(n + k)

Why don't we always use counting sort? Depends on range k of elements.

Counting Sort: Time complexity analysis

Lemma

Provided that all elements of $A[1 \dots n]$ are integers from 1 to M, $\mathrm{CountSort}(A)$ sorts A in time $O(n + M)$.

Counting Sort: Time complexity analysis

Lemma

Provided that all elements of $A[1 \ldots n]$ are integers from 1 to M, $\mathrm{CountSort}(A)$ sorts A in time $O(n + M)$.

Remark

If $M = O(n)$, then the running time is $O(n)$.

Counting Sort Review

- **Assumption:** input taken from **small** set of **numbers** of size k
- Basic idea:
  - Count number of elements less than you for each element.
  - This gives the position of that number – similar to selection sort.
- Pro's:
  - Fast … $O(n+k)$
  - Simple to code
- Con's:
  - Does not sort in place.
  - Elements must be integers. ***countable***
  - Requires $O(n+k)$ extra storage.

Radix Sort

- Intuitively, you might sort on the <u>most significant digit</u>, then the second msd, etc.
- Problem: lots of intermediate piles of information to keep track of
- Key idea: sort the <u>*least* significant digit</u> first

```
RadixSort(A, d)
    for i=1 to d
        StableSort(A) on digit i
```

Radix Sort Correctness

- Sketch of an inductive proof of correctness (induction on the number of passes):
  - Assume lower-order digits {*j: j<i* }are sorted
  - Show that sorting next digit *i* leaves array correctly sorted
    - If <u>two digits at position *i* are different</u>, ordering numbers by that digit is <u>correct</u> (lower-order digits irrelevant)
    - If they are the <u>same</u>, numbers are already sorted on the lower-order digits. Since we use a <u>stable sort</u>, the numbers stay in the right order

Radix Sort

- *What sort is used to sort on digits?*
- **Counting** sort is obvious choice:
  - Sort n numbers on digits that range from $1..k$
  - Time: $O(n + k)$
- Each pass over n numbers with d digits takes time $O(n+k)$, so total time $O(dn+dk)$

Radix Sort Review

- **Assumption:** input has *d* digits ranging from 0 to *k*
- Basic idea:
  - Sort elements by digit starting with *least* significant
  - Use a stable sort (like counting sort) for each stage
- Pro's:
  - Fast
  - Simple to code
- Con's:
  - Doesn't sort in place

Quiz 1

- What is the main characteristics of the sorting algorithm used in the Radix sort?
- Can we use Quick sort to improve the original implementation?

Discussion

- How we could improve the **Divide and Conquer** approach?

Summary

- Merge sort uses the divide-and-conquer strategy to sort an n-element array in time $O(n \log n)$.
- No comparison based algorithm can do this (asymptotically) faster.
- One can do faster if something is known about the input array in advance (e.g., it contains small integers).