CSC 4520/6520
Design & Analysis of Algorithms

MIDTERM 2 REVİEW

Abdullah Bal, PhD

Georgia State University

CSC 4520/6520
Design & Analysis of Algorithms

CHAPTER 4: DECREASE-AND-CONQUER
(INSERTION SORT , TOPOLOGICAL SORTING)

Abdullah Bal, PhD
Spring 2024

Decrease-and-Conquer

1. Reduce problem instance to smaller instance of the same problem
2. Solve smaller instance
3. Extend solution of smaller instance to obtain solution to original instance

Decrease-and-Conquer

- Can be implemented either top-down (recursive implementation) or bottom-up (nonrecursive):
- The bottom-up variation is usually implemented iteratively, starting with a solution to the smallest instance of the problem; it is called sometimes the incremental approach.
- There are three major variations of decrease-and-conquer:
  - Decrease by a constant
  - Decrease by a constant factor
  - Variable size decrease

Types of Decrease and Conquer

Decrease by a constant (usually by 1):

- Insertion sort
- Topological sorting
- Algorithms for generating permutations, subsets

Decrease by a constant factor (usually by half)

- Binary search and bisection method
- Exponentiation by squaring
- Multiplication à la russe

Variable-size decrease

- Euclid's algorithm
- Selection by partition
- Nim-like games

Insertion Sort

Insertion Sort

- To sort array A[0..*n*-1], sort A[0..*n*-2] recursively and then insert A[*n*-1] in its proper place among the sorted A[0..*n*-2]
- Usually implemented bottom up (nonrecursively)
- Example: Sort 6, 4, 1, 8, 5

6 | 4 1 8 5
4 6 | 1 8 5
1 4 6 | 8 5
1 4 6 8 | 5
1 4 5 6 8

Analysis of Insertion Sort

- The basic operation of the algorithm is the key comparison $A[j] > v$.
- Why not $j \geq 0$?
- Because it is almost certainly faster than the former in an actual computer implementation.
- The number of key comparisons in this algorithm obviously depends on the nature of the input.

Analysis of Insertion Sort

- In the worst case, A[j] > v is executed the largest number of times, i.e., for every
- j = i − 1, . . . , 0.
- For the worst-case input, we get A[0] > A[1] (for i = 1),

  A[1] > A[2] (for i = 2), . . . , A[n − 2] > A[n − 1] (for i = n − 1).
- In other words, the worst-case input is an array of strictly decreasing values. The number of key comparisons for such an input is

$$C_{worst}(n) = \sum_{i=1}^{n-1} \sum_{j=0}^{i-1} 1 = \sum_{i=1}^{n-1} i = \frac{(n-1)n}{2} \in \Theta(n^2).$$

- In the worst case, insertion sort makes exactly the same number of comparisons as selection sort

Analysis of Insertion Sort

- In the best case, the comparison A[j]>v is executed only once on every iteration of the outer loop.
- It happens if and only if A[i − 1] ≤ A[i] for every i = 1, . . . , n − 1, i.e., if the input array is already sorted in nondecreasing order.
- Thus, for sorted arrays, the number of key comparisons is

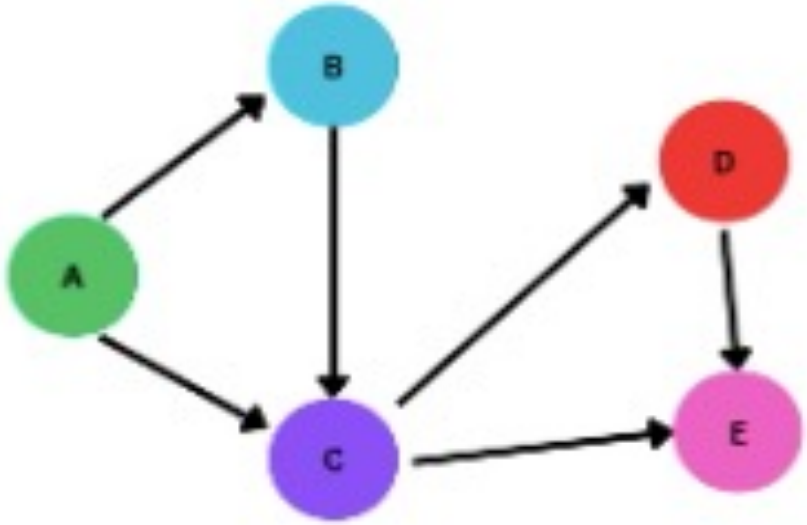
$$C_{best}(n) = \sum_{i=1}^{n-1} 1 = n - 1 \in \Theta(n).$$

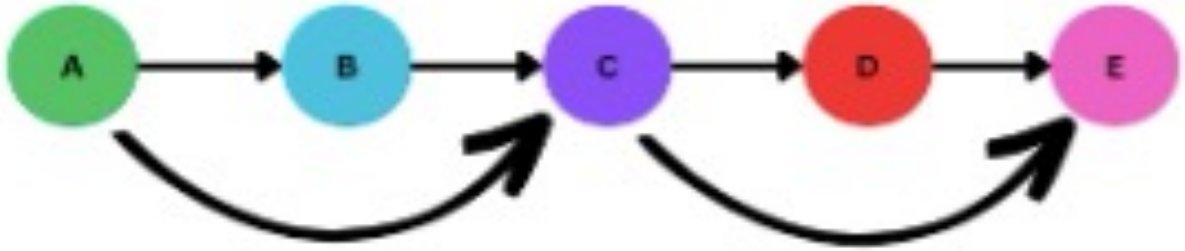
Analysis of Insertion Sort

- A rigorous analysis of the algorithm's average-case efficiency is based on investigating the number of element pairs that are out of order.
- It shows that on randomly ordered arrays, insertion sort makes on average half as many comparisons as on decreasing arrays, i.e.,

$$C_{avg}(n) \approx \frac{n^2}{4} \in \Theta(n^2).$$

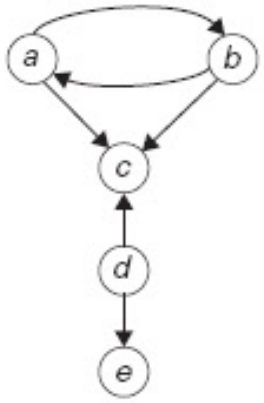
Topological Sorting



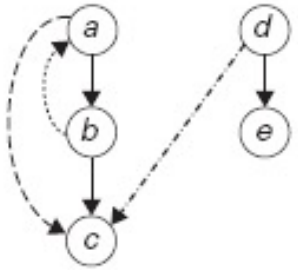


Topological Sorting

- A ***directed graph***, or ***digraph*** for short, is a graph with directions specified for all its edges.
- There are only two notable differences between undirected and directed graphs in representing them:
  1) The adjacency matrix of a directed graph does not have to be symmetric;
  2) An edge in a directed graph has just one (not two) corresponding nodes in the digraph's adjacency lists.
-



Digraph.

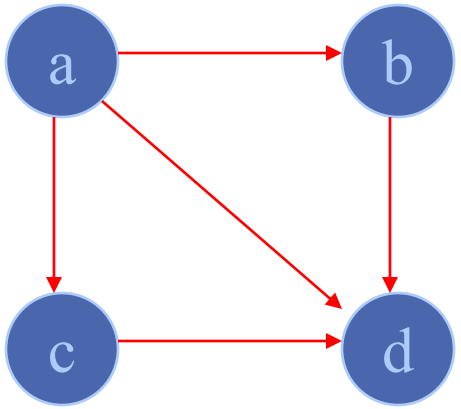


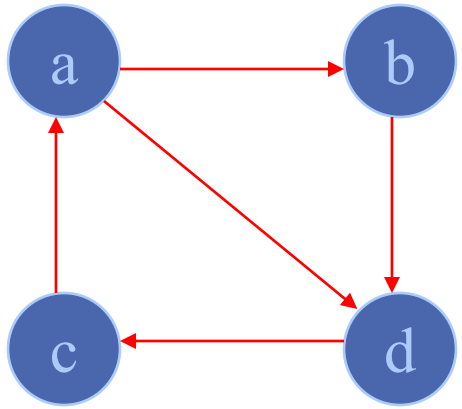
DFS forest of the digraph for the DFS traversal started at a.

Dags and Topological Sorting

- A *dag*: a directed acyclic graph, i.e. a directed graph with no (directed) cycles
- If a DFS forest of a digraph has no back edges, the digraph is a ***dag***, an acronym for ***directed acyclic graph***.

a dag





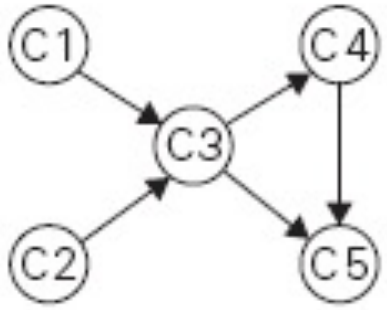
not a dag

Dags and Topological Sorting

- Arise in modeling many problems that involve prerequisite constraints (construction projects, document version control, course selection etc.)
- **Topological Sorting :** Vertices of a dag can be linearly ordered so that for every edge its starting vertex is listed before its ending vertex.
- Being a dag is also a necessary condition for topological sorting be possible.

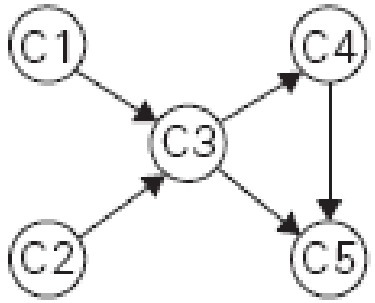
Example: Course Selection

- Consider a set of five required courses {C1, C2, C3, C4, C5} a part-time student has to take in some degree program.
- The courses can be taken in any order as long as the following course prerequisites are met: C1 and C2 have no prerequisites, C3 requires C1 and C2, C4 requires C3, and C5 requires C3 and C4.
- The situation can be modeled by a digraph in which vertices represent courses and directed edges indicate prerequisite requirements.
- The student can take only one course per term. In which order should the student take the courses?



The First Algorithm: DFS

- In terms of this digraph, the question is whether we can list its vertices in such an order that for every edge in the graph, the vertex where the edge starts is listed before the vertex where the edge ends.
- This problem is called ***topological sorting***.



Stack

$C5_1$
$C4_2$
$C3_3$
$C1_4$ $C2_5$

The topologically sorted list:

C2 C1→C3→C4→C5

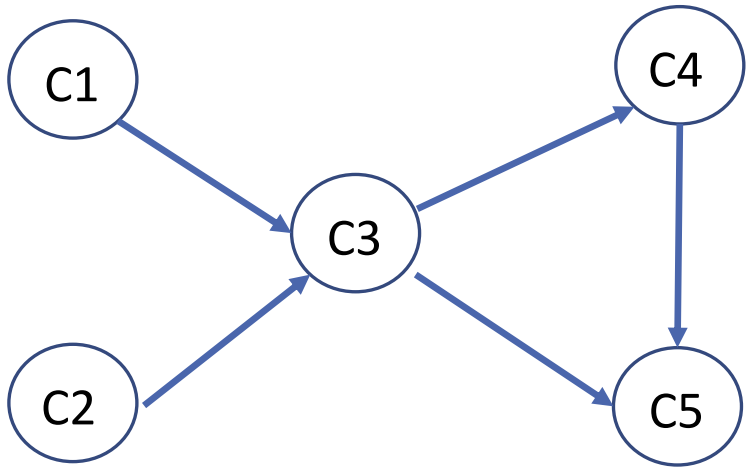
- DFS traversal stack with the subscript numbers indicating the popping- off order.

The second algorithm: Source-removal algorithm

- The second algorithm is based on a direct implementation of the decrease-and-conquer technique (by one)
- Repeatedly, identify in a remaining digraph a ***source***, which is a vertex with no incoming edges (Indegree=0), and delete it along with all the edges outgoing from it.
- If there are several sources, break the tie arbitrarily.
- If there are none, stop because the problem cannot be solved.
- The order in which the vertices are deleted yields a solution to the topological sorting problem.

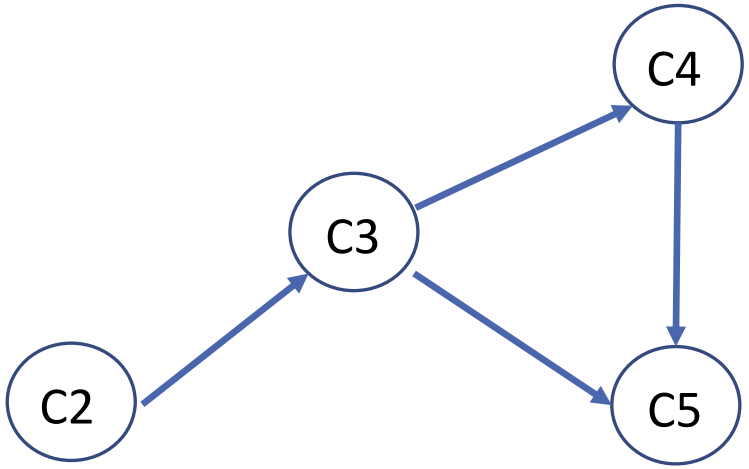
Example 1: Source-removal algorithm

- Vertices which have no incoming edges?
- Indegree=0



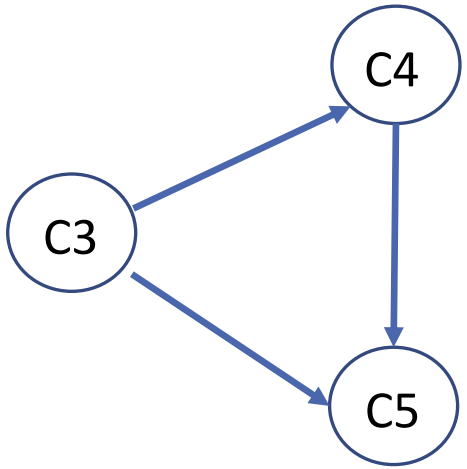
The solution: ?

Example 1: Source-removal algorithm



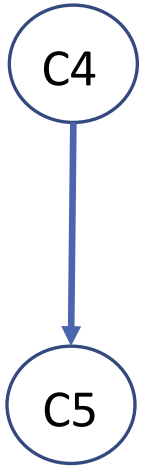
The solution: C1

Example 1: Source-removal algorithm



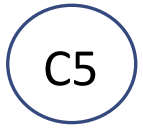
The solution: C1 C2

Example 1: Source-removal algorithm



The solution: C1 C2 C3

Example 1: Source-removal algorithm

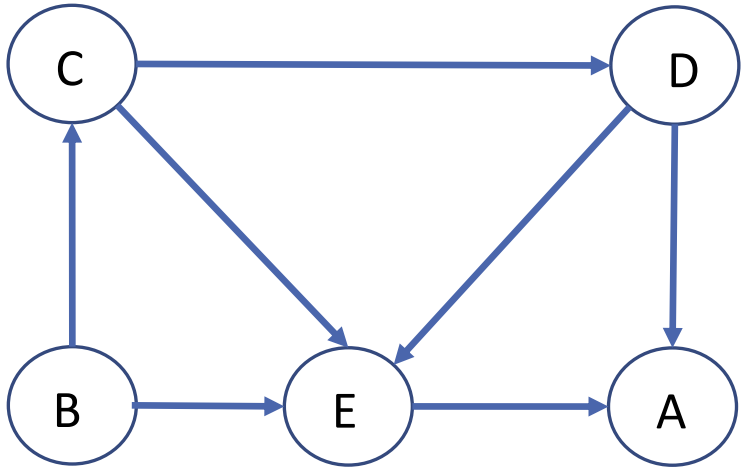


The solution: C1 C2 C3 C4

Example 1: Source-removal algorithm

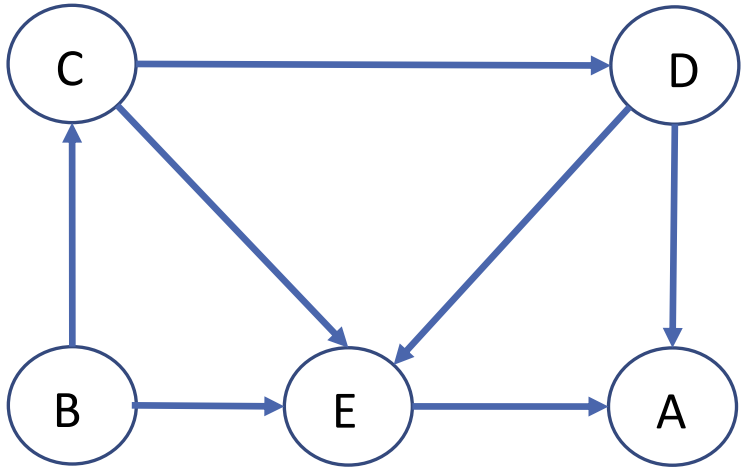
The source-removal algorithm solution: C1 C2 C3 C4 C5

Example 2: Source-removal algorithm



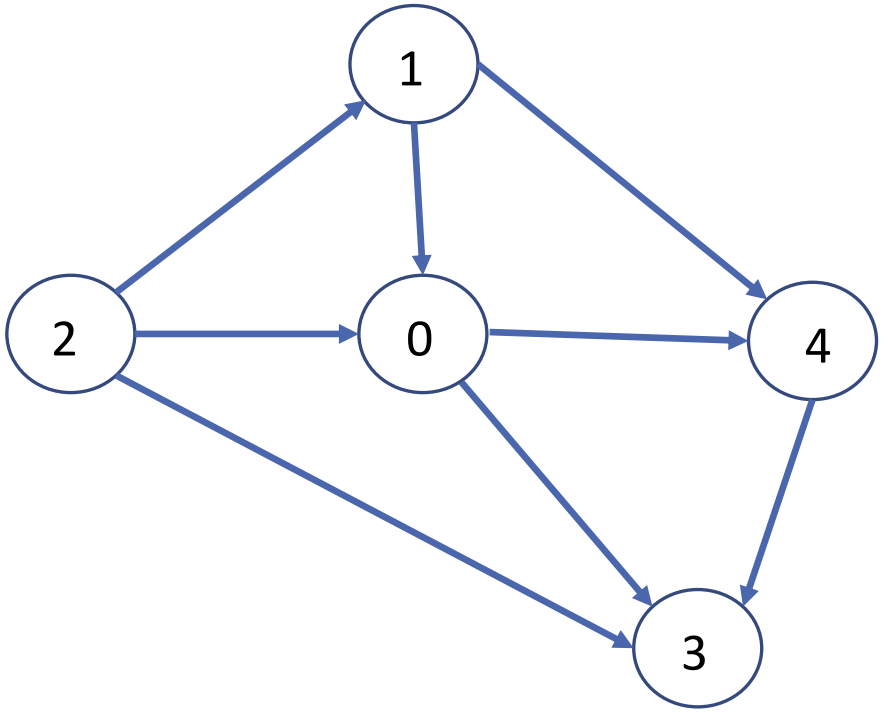
Topological sorting: ?

Example 2: Source-removal algorithm



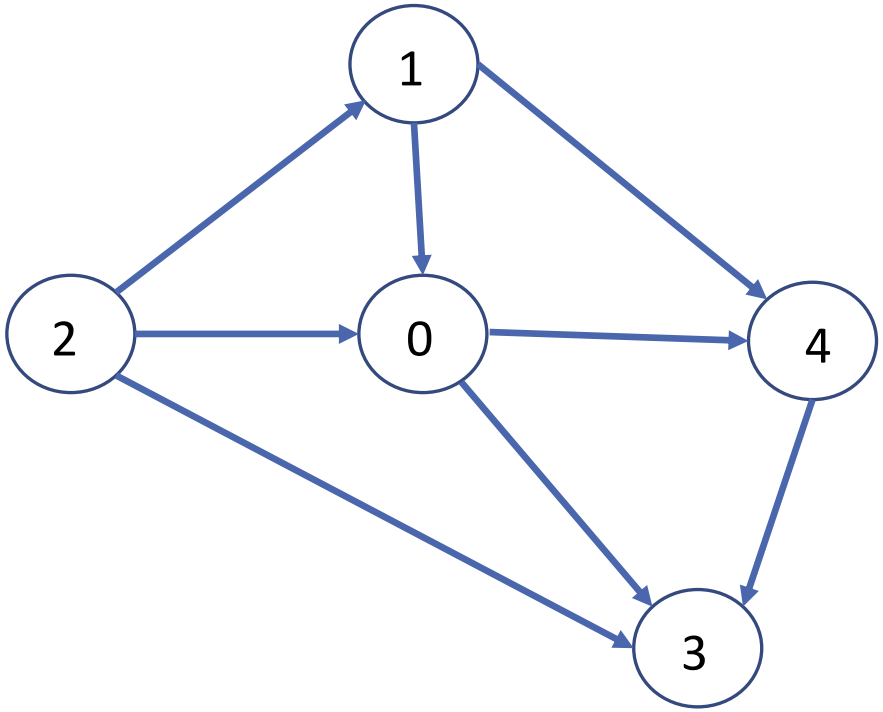
Topological sorting: B C D E A

Example 3: Source-removal algorithm



Topological sorting: ?

Example 3: Source-removal algorithm



Topological sorting: 2 1 0 4 3

DFS vs. Source-Removal Algorithm

- Note that the solution obtained by the source-removal algorithm is different from the one obtained by the DFS-based algorithm.
- Both of them are correct, of course; the topological sorting problem may have several alternative solutions.

The DFS solution: C2 C1 C3 C4 C5

The source-removal algorithm solution: C1 C2 C3 C4 C5

Georgia State University

CSC 4520/6520
Design & Analysis of Algorithms

CHAPTER 4: DECREASE-AND-CONQUER
(GENERATING COMBINATORIAL OBJECTS)

Abdullah Bal, PhD
Spring 2024

Algorithms for Generating Combinatorial Objects

- The most important types of combinatorial objects are
  - permutations
  - combinations
  - subsets of a given set
- They typically arise in problems that require a consideration of different choices.
- The number of combinatorial objects typically grows exponentially or even faster as a function of the problem size.
- Our primary interest here lies in algorithms for generating combinatorial objects, not just in counting them.

1-Generating Permutations: Minimal-change

- If $n = 1$ return 1;
- Otherwise, generate recursively the list of all permutations of

  1 2…n-1
- Then insert *n* into each of those permutations by starting with inserting *n* into 1 2...*n*-1 by moving right to left.

1-Generating Permutations: Minimal-change

Example: n=3

start	1		
insert 2 into 1 right to left	12	21	
insert 3 into 12 right to left	123	132	312
insert 3 into 21 right to left	213	231	321

2-Generating Permutations: Johnson-Trotter Algorithm

- It is possible to get the same ordering of permutations of n elements without explicitly generating permutations for smaller values of n.
- It can be done by associating a direction with each element k in a permutation.
- We indicate such a direction by a small arrow written above the element in question, e.g.,

$$\overset{\rightarrow}{3}\quad\overset{\leftarrow}{2}\quad\overset{\rightarrow}{4}\quad\overset{\leftarrow}{1}$$

- The element k is said to be ***mobile*** in such an arrow-marked permutation if its arrow points to a smaller number adjacent to it.
- 3 and 4 are mobile while 2 and 1 are not.
- Using the notion of a mobile element, we can give the following description of the ***Johnson-Trotter algorithm*** for generating permutations.

2- Johnson-Trotter Algorithm (Cont.)

ALGORITHM *JohnsonTrotter*(n)

//Implements Johnson-Trotter algorithm for generating permutations
//Input: A positive integer n
//Output: A list of all permutations of $\{1, \ldots, n\}$
initialize the first permutation with $\overleftarrow{1}\ \overleftarrow{2} \ldots \overleftarrow{n}$
while the last permutation has a mobile element **do**
    find its largest mobile element k
    swap k with the adjacent element k's arrow points to
    reverse the direction of all the elements that are larger than k
    add the new permutation to the list

2- Johnson-Trotter Algorithm (Cont.)

- An application of this algorithm for n = 3, with the largest mobile element shown in bold:

$\overleftarrow{1}\ \overleftarrow{2}\ \overleftarrow{\mathbf{3}}\quad \overleftarrow{1}\ \overleftarrow{\mathbf{3}}\ \overleftarrow{2}\quad \overleftarrow{3}\ \overleftarrow{1}\ \overleftarrow{\mathbf{2}}\quad \overrightarrow{\mathbf{3}}\ \overleftarrow{2}\ \overleftarrow{1}\quad \overleftarrow{2}\ \overrightarrow{\mathbf{3}}\ \overleftarrow{1}\quad \overleftarrow{2}\ \overleftarrow{1}\ \overrightarrow{3}.$

2- Johnson-Trotter Algorithm (Cont.)

- This algorithm is one of the most efficient for generating permutations
- It can be implemented to run in time proportional to the number of permutations, i.e., in $\Theta(n!)$.
- It is horribly slow for all but very small values of n;
- This is not the algorithm's "fault" but rather the fault of the problem
- It simply asks to generate too many items.

Generating Subsets

- The knapsack problem, which asks to find the most valuable subset of items that fits a knapsack of a given capacity.
- The exhaustive-search approach to solving this problem discussed there was based on generating all subsets of a given set of items.
- We discuss algorithms for generating all 2^n subsets of an abstract set

$$A = \{a_1, \ldots, a_n\}.$$

- Mathematicians call the set of all subsets of a set its ***power set***.

1- Generating Subsets: Decrease-by-one

- The decrease-by-one idea is immediately applicable to this problem.
- All subsets of A = $\{a_1, \ldots, a_n\}$ can be divided into two groups: those that do not contain a_n and those that do.
- The former group is nothing but all the subsets of $\{a_1, \ldots, a_{n-1}\}$, while each and every element of the latter can be obtained by adding a_n to a subset of $\{a_1, \ldots, a_{n-1}\}$.
- Thus, once we have a list of all subsets of $\{a_1,\ldots,a_{n-1}\}$, we can get all the subsets of $\{a_1, \ldots, a_n\}$ by adding to the list all its elements with an put into each of them.

1- Generating Subsets: Decrease-by-one

- An application of this algorithm (**bottom-up**) to generate all subsets of {a1, a2, a3} is illustrated as follow:

n	subsets							
0	$\varnothing$							
1	$\varnothing$	$\{a_1\}$						
2	$\varnothing$	$\{a_1\}$	$\{a_2\}$	$\{a_1, a_2\}$				
3	$\varnothing$	$\{a_1\}$	$\{a_2\}$	$\{a_1, a_2\}$	$\{a_3\}$	$\{a_1, a_3\}$	$\{a_2, a_3\}$	$\{a_1, a_2, a_3\}$

2- Generating Subsets: Bit Strings

- A convenient way of solving the problem directly is based on a one-to-one correspondence between all 2^n subsets of an n element set $A = \{a1, \ldots, a_n\}$ and all 2^n bit strings $b1, \ldots, b_n$ of length n.
- The easiest way to establish such a correspondence is to assign to a subset the bit string in which $b_i = 1$ if a_i belongs to the subset and $b_i = 0$ if a_i does not belong to it.

2- Generating Subsets: Bit Strings

- For example, the bit string 000 will correspond to the empty subset of a three-element set, 111 will correspond to the set itself, i.e., {a1, a2, a3}, and 110 will represent {a1, a2}.
- For the case of n = 3, we obtain

bit strings	000	001	010	011	100	101	110	111
subsets	$\varnothing$	$\{a_3\}$	$\{a_2\}$	$\{a_2, a_3\}$	$\{a_1\}$	$\{a_1, a_3\}$	$\{a_1, a_2\}$	$\{a_1, a_2, a_3\}$

Georgia State University

CSC 4520/6520
Design & Analysis of Algorithms

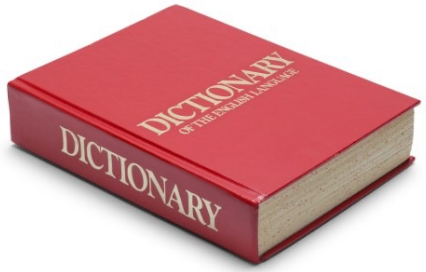
CHAPTER 4: DECREASE-AND-CONQUER

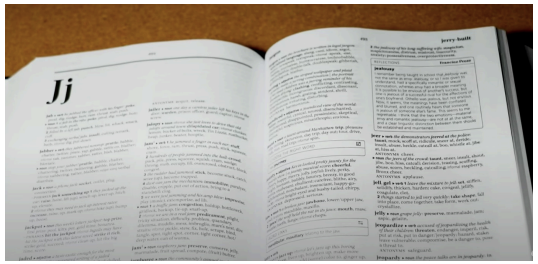
(DECREASE-BY-CONSTANT-FACTOR & VARIABLE SIZE DECREASE)

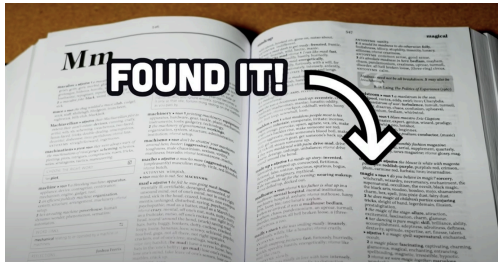
Abdullah Bal, PhD
Spring 2024

2- Decrease-by-Constant-Factor Algorithms

- In this variation of decrease-and-conquer, instance size is reduced by the same factor (typically, 2)
- Decrease-by-a-constant-factor algorithms usually run in logarithmic time, and being very efficient,
- A reduction by a factor other than two is especially rare





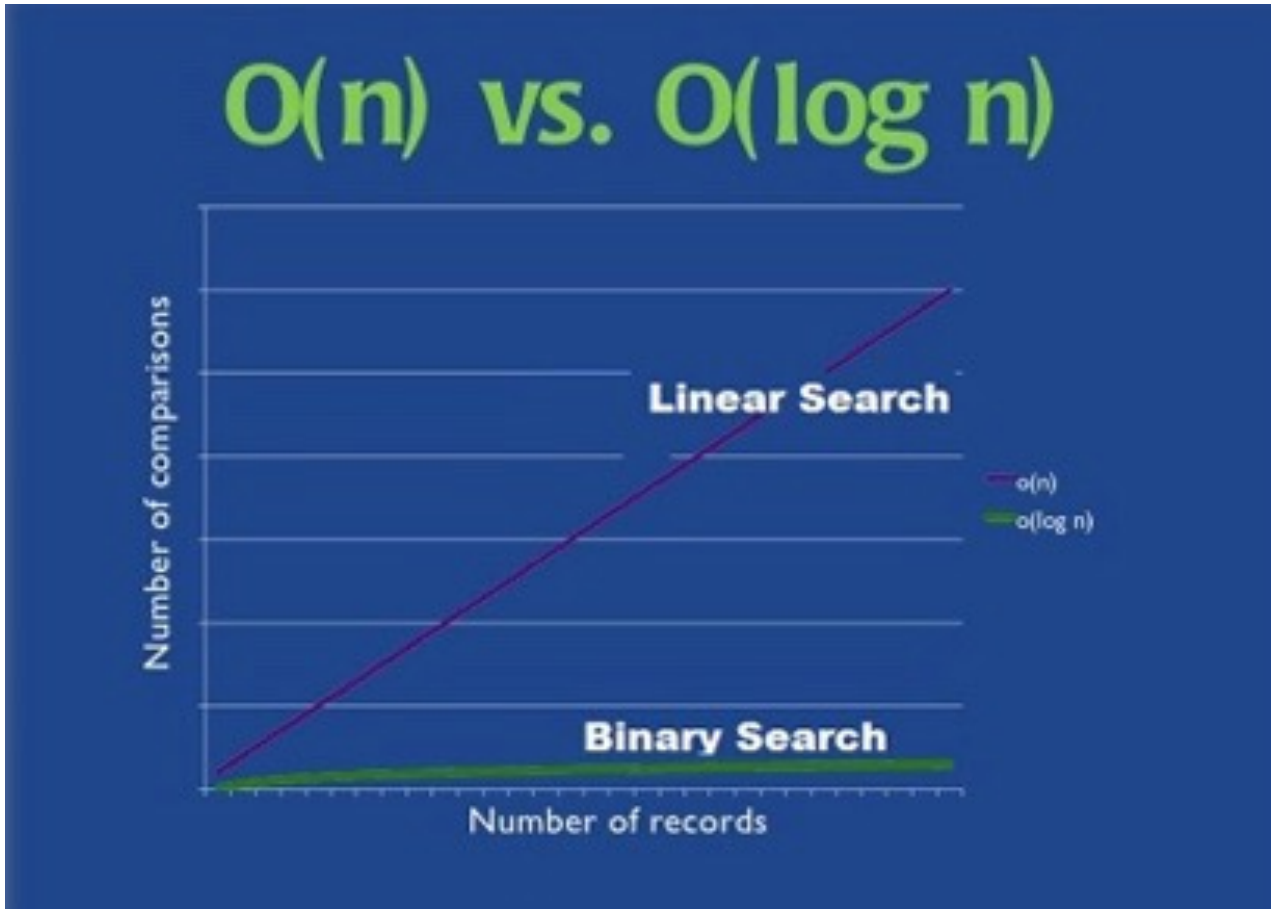


2- Decrease-by-Constant-Factor Algorithms

Examples:

- Binary search and the method of bisection
- Exponentiation by squaring
- Multiplication à la russe (Russian peasant method)
- Fake-coin puzzle

Example 1: Binary Search



Example 1: Binary Search

- Very efficient algorithm for searching in sorted array:

- If K = A[m], stop (successful search);

  otherwise,

- Continue searching by the same method

  in A[0..m-1] if K < A[m]

  and

  in A[m+1..n-1] if K > A[m]

$$\underbrace{A[0] \ldots A[m-1]}_{\substack{\text{search here if} \\ K < A[m]}} \quad \begin{matrix} K \\ \updownarrow \\ A[m] \end{matrix} \quad \underbrace{A[m+1] \ldots A[n-1]}_{\substack{\text{search here if} \\ K > A[m]}}.$$

Example 1: Binary Search

ALGORITHM *BinarySearch*$(A[0..n-1], K)$

```
//Implements nonrecursive binary search
//Input: An array A[0..n − 1] sorted in ascending order and
//       a search key K
//Output: An index of the array's element that is equal to K
//        or −1 if there is no such element
l ← 0;    r ← n − 1
while l ≤ r do
    m ← ⌊(l + r)/2⌋
    if K = A[m] return m
    else if K < A[m] r ← m − 1
    else l ← m + 1
return −1
```

Example 1: Binary Search

- As an example, let us apply binary search to searching for K = 70 in the array

3	14	27	31	39	42	55	70	74	81	85	93	98

- The iterations of the algorithm are given in the following table:

index	0	1	2	3	4	5	6	7	8	9	10	11	12
value	3	14	27	31	39	42	55	70	74	81	85	93	98
iteration 1	*l*						*m*						*r*
iteration 2								*l*		*m*			*r*
iteration 3								*l*,*m*	*r*				

- Though binary search is clearly based on a recursive idea, it can be easily implemented as a nonrecursive algorithm, too.

Analysis of Binary Search

- The worst-case inputs include all arrays that do not contain a given search key, as well as some successful searches.
- Since after one comparison the algorithm faces the same situation but for an array half the size, we get the following recurrence relation for $C_{worst}(n)$:

$$C_{worst}(n) = C_{worst}(\lfloor n/2 \rfloor) + 1 \quad \text{for } n > 1, \quad C_{worst}(1) = 1.$$

- Using backward substitution, we obtain

$$C_{worst}(n) = \lfloor \log_2 n \rfloor + 1$$

- Time complexity of the binary search O(log n).

Analysis of Binary Search

- A sophisticated analysis shows that the average number of key comparisons made by binary search is only slightly smaller than that in the worst case:

$$C_{avg}(n) \approx \log_2 n.$$

3- Variable-Size-Decrease Algorithms

- Instance size reduction varies from one iteration to another
- Examples:
  - Euclid's algorithm for greatest common divisor
  - Partition-based algorithm for selection problem
  - Interpolation search
  - Some algorithms on binary search trees
  - Nim and Nim-like games

Example 1: Euclid's Algorithm

- Euclid's algorithm is based on repeated application of equality

$$\gcd(m, n) = \gcd(n, m \bmod n)$$

Ex.: gcd(80,44) = gcd(44,36) = gcd(36, 12) = gcd(12,0) = 12

- One can prove that the size, measured by the second number,decreases at least by half after two consecutive iterations.
- Hence, $T(n) \in O(\log n)$

Example 2: Selection Problem

- Finding the k^{th} smallest element in a list of n numbers.
- This number is called the k^{th} ***order statistic***.
- For k = 1 or k = n, we can simply scan the list in question to find the smallest or largest element, respectively.
- A more interesting case of this problem is for k = ⌈n/2⌉, which asks to find an element that is not larger than one half of the list's elements and not smaller than the other half.
- This middle value is called the ***median***, and it is one of the most important notions in mathematical statistics.

Example 2: Selection Problem

- We can find the k^{th} smallest element in a list by sorting the list first and then selecting the k^{th} element in the output of a sorting algorithm.
- The time of such an algorithm is determined by the efficiency of the sorting algorithm used.
- Thus, with a fast-sorting algorithm such as mergesort, the algorithm's efficiency is in O(n log n).

Example 2: Selection Problem

median: $k = \lceil n/2 \rceil$

Example: 4, 1, 10, 9, 7, 12, 8, 2, 15 median = ?

- The median is used in statistics as a measure of an average value of a sample.
- In fact, it is a better (more robust) indicator than the mean, which is used for the same purpose.

Partitioning Algorithms for the Selection Problem

- Sorting the entire list is most likely overkill since the problem asks not to order the entire list but just to find its k^{th} smallest element.
- We can take advantage of the idea of ***partitioning*** a given list around some value p of, say, its first element.
- In general, this is a rearrangement of the list's elements so that the left part contains all the elements smaller than or equal to p, followed by the ***pivot*** p itself, followed by all the elements greater than or equal to p.

Two Partitioning Algorithms

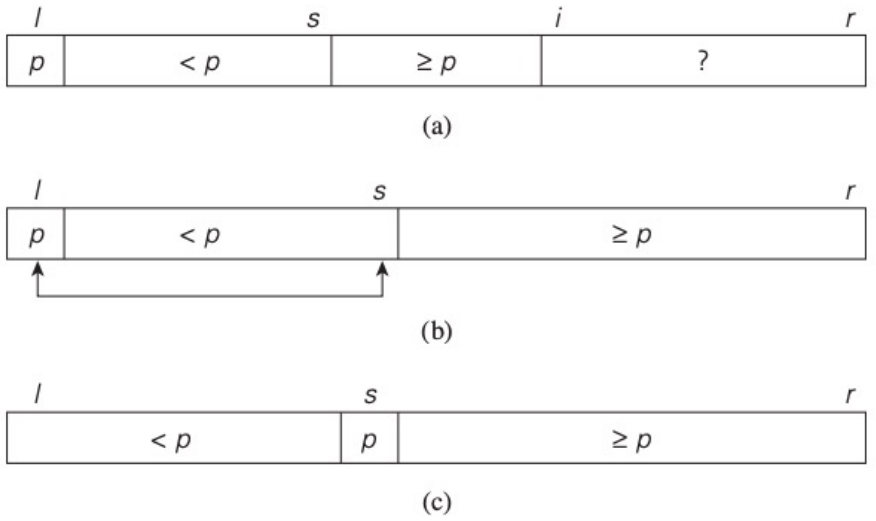
- There are two principal ways to partition an array:
  - One-directional scan (Lomuto's partitioning algorithm)
  - Two-directional scan (Hoare's partitioning algorithm)

Lomuto's Partitioning Algorithm

- It is helpful to think of an array—or, more generally, a subarray under consideration as composed of three contiguous segments.

$$A[l.....r] \qquad (0 \le l \le r \le n - 1)$$

- Listed in the order they follow pivot p, they are as follows:
  - a segment with elements known to be smaller than p,
  - the segment of elements known to be greater than or equal to p,
  - the segment of elements yet to be compared to p.
- Note that the segments can be empty; for example, it is always the case for the first two segments before the algorithm starts.



Lomuto's Partitioning Algorithm

- Starting with i = l + 1, the algorithm scans the subarray A[l….r] left to right, maintaining this structure until a partition is achieved.
- On each iteration, it compares the first element in the unknown segment (pointed to by the scanning index i) with the pivot p.
- If A[i] ≥ p, i is simply incremented to expand the segment of the elements greater than or equal to p while shrinking the unprocessed segment.
- If A[i] < p, it is the segment of the elements smaller than p that needs to be expanded.
- This is done by incrementing s, the index of the last element in the first segment, swapping A[i] and A[s], and then incrementing i to point to the new first element of the shrunk unprocessed segment.
- After no unprocessed elements remain, the algorithm swaps the pivot with A[s] to achieve a partition being sought.

Lomuto's Partitioning Algorithm

ALGORITHM *LomutoPartition*$(A[l..r])$

```
//Partitions subarray by Lomuto's algorithm using first element as pivot
//Input: A subarray A[l..r] of array A[0..n − 1], defined by its left and right
//        indices l and r (l ≤ r)
//Output: Partition of A[l..r] and the new position of the pivot
p ← A[l]
s ← l
for i ← l + 1 to r do
    if A[i] < p
        s ← s + 1;   swap(A[s], A[i])
swap(A[l], A[s])
return s
```

Tracing Lomuto's Partioning Algorithm

s	*i*							
4	1	10	8	7	12	9	2	15
	s	*i*						
4	1	10	8	7	12	9	2	15
	s						*i*	
4	1	10	8	7	12	9	2	15
		s						*i*
4	1	2	8	7	12	9	10	15
		s						
4	1	2	8	7	12	9	10	15
2	1	4	8	7	12	9	10	15

$p \leftarrow A[l]$
$s \leftarrow l$
for $i \leftarrow l + 1$ **to** r **do**
    if $A[i] < p$
        $s \leftarrow s + 1$; swap($A[s]$, $A[i]$)
swap($A[l]$, $A[s]$)
return s

Quickselect (Partition-based Algorithm)

- Let us assume that the list is implemented as an array whose elements are indexed starting with a 0, and let s be the partition's split position, i.e., the index of the array's element occupied by the pivot after partitioning.
- If $s = k - 1$, pivot p itself is obviously the k^{th} smallest element, which solves the problem.
- If $s > k - 1$, the k^{th} smallest element in the entire array can be found as the k^{th} smallest element in the left part of the partitioned array.
- And if $s < k - 1$, it can be found as the $(k - s)$th smallest element in its right part.
- Thus, if we do not solve the problem outright, we reduce its instance to a smaller one, which can be solved by the same approach, i.e., recursively.
- This algorithm is called ***quickselect***.

Tracing Quickselect

- Apply the partition-based algorithm to find the median of the following list of nine numbers: 4, 1, 10, 8, 7, 12, 9, 2, 15.
- Find the 5^{th} (k = [9/2] = 5) smallest element in the array.

0	1	2	3	4	5	6	7	8
s	*i*							
4	1	10	8	7	12	9	2	15
	s	*i*						
4	1	10	8	7	12	9	2	15
	s						*i*	
4	1	10	8	7	12	9	2	15
		s					*i*	
4	1	2	8	7	12	9	10	15
		s						*i*
4	1	2	8	7	12	9	10	15
2	1	**4**	8	7	12	9	10	15

Tracing Quickselect

- Since s = 2 is smaller than k − 1 = 4, we proceed with the right part of the array:

0	1	2	3	4	5	6	7	8
			s	*i*				
			8	7	12	9	10	15
				s	*i*			
			8	7	12	9	10	15
				s				*i*
			8	7	12	9	10	15
			7	**8**	12	9	10	15

- Now s = k − 1 = 4, and hence we can stop.
- The found median is 8, which is greater than 2, 1, 4, and 7 but smaller than 12, 9, 10, and 15.

Quickselect

ALGORITHM *Quickselect*$(A[l..r], k)$

//Solves the selection problem by recursive partition-based algorithm
//Input: Subarray $A[l..r]$ of array $A[0..n-1]$ of orderable elements and
// integer k $(1 \le k \le r - l + 1)$
//Output: The value of the kth smallest element in $A[l..r]$
$s \leftarrow$ *LomutoPartition*$(A[l..r])$ //or another partition algorithm
if $s = k - 1$ **return** $A[s]$
else if $s > l + k - 1$ *Quickselect*$(A[l..s-1], k)$
else *Quickselect*$(A[s+1..r], k - 1 - s)$

Efficiency of Quickselect

- Partitioning an n-element array always requires $n - 1$ key comparisons.
- If it produces the split that solves the selection problem without requiring more iterations, then for this best case we obtain $C_{best}(n) = n - 1 \in \Theta(n)$.

Efficiency of Quickselect

- Unfortunately, the algorithm can produce an extremely unbalanced partition of a given array, with one part being empty and the other containing n − 1 elements.
- In the worst case, this can happen on each of the n − 1 iterations.

$$C_{worst}(n) = (n - 1) + (n - 2) + \ldots + 1 = (n - 1)n/2 \in \Theta(n^2)$$

which compares poorly with the straightforward sorting-based approach.

- In fact, computer scientists have discovered a more sophisticated way of choosing a pivot in quickselect that guarantees linear time even in the worst case ($\Theta(n)$) but it is too complicated to be recommended for practical applications.

CSC 4520/6520
Design & Analysis of Algorithms

CHAPTER 5: DIVIDE AND CONQUER

(MERGE SORT-QUICK SORT)

Abdullah Bal, PhD
Spring 2024

Divide-and-Conquer

The most-well known algorithm design strategy:

1. Divide instance of problem into two or more smaller instances

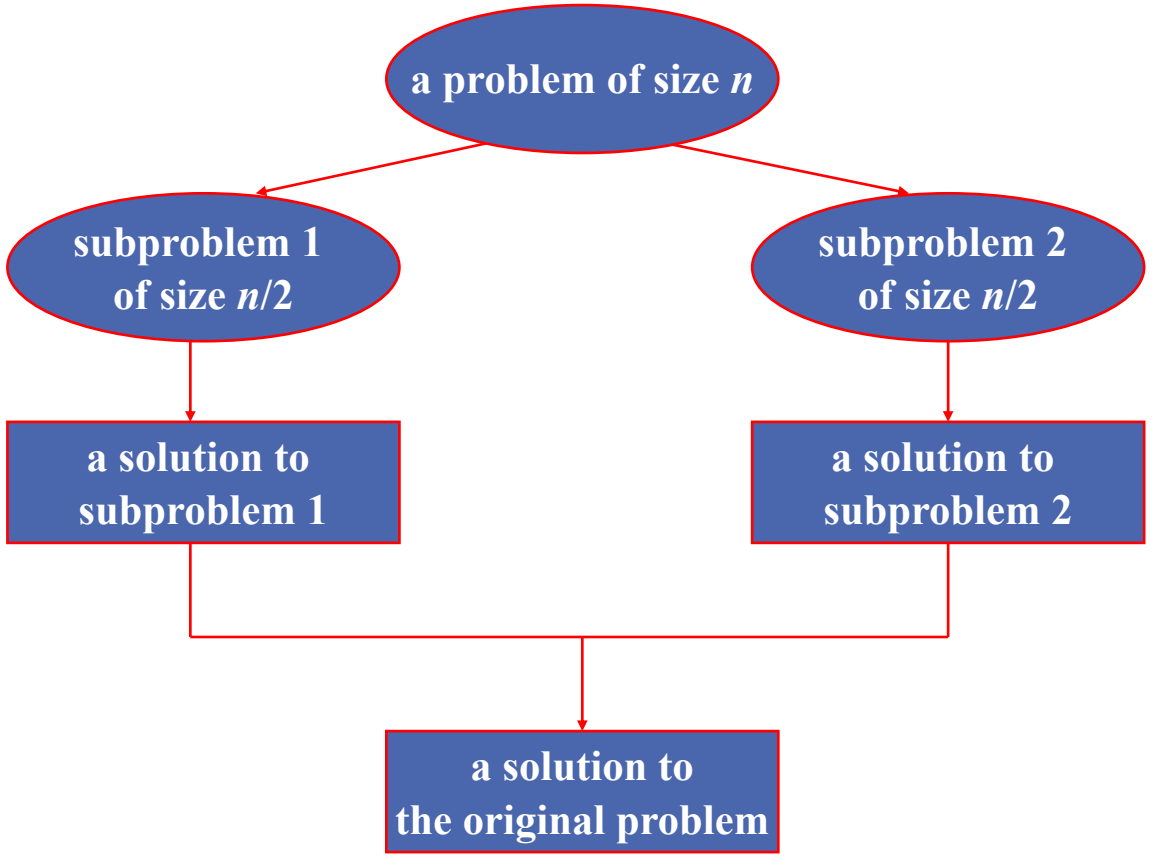
2. Solve smaller instances recursively

3. Obtain solution to original (larger) instance by combining these solutions

Divide-and-Conquer

- The divide-and-conquer approach yields some of the most important and efficient algorithms in computer science.
- The divide-and-conquer technique is ideally suited for parallel computations, in which each subproblem can be solved simultaneously by its own processor.

Divide-and-Conquer Technique (cont.)



Divide-and-Conquer Examples

- Sorting: Merge Sort and Quick Sort
- Binary tree traversals
- Multiplication of large integers
- Matrix multiplication: Strassen’s algorithm
- Closest-pair and convex-hull algorithms

Divide-and-Conquer Analysis

- In the most typical case of divide-and-conquer a problem's instance of size n is divided into two instances of size n/2.
- More generally, an instance of size *n* can be divided into *b* instances of size *n/b*, with *a* of them needing to be solved.

  (*a* and *b* are constants; $a \geq 1$ and $b > 1$.)
- Assuming that size *n* is a power of *b* to simplify our analysis, we get the following recurrence for the running time T (n):

$$T(n)=a.T(n/b)+f(n)$$

Divide-and-Conquer Analysis

$$T(n)=a.T(n/b)+f(n)$$

where f (n) is a function that accounts for the time spent on dividing an instance of size n into instances of size n/b and combining their solutions.

- This recursive formula is called the **general divide-and-conquer recurrence**.
- T (n) depends on the values of the constants *a* and *b* and the order of growth of the function f (n).

Master theorem

- General divide-and-conquer recurrence formula:

$$T(n)=a.T(n/b)+f(n)$$

- The efficiency analysis of many divide-and-conquer algorithms is greatly simplified by the following Master theorem:

- If $f(n) \in \Theta(n^d)$ where $d \geq 0$, then

If $a < b^d$, $T(n) \in \Theta(n^d)$

If $a = b^d$, $T(n) \in \Theta(n^d \log n)$

If $a > b^d$, $T(n) \in \Theta(n^{\log_b a})$

- Analogous results hold for the O and Ω notations, too.

Master theorem Examples

Examples: $T(n) = 4T(n/2) + n \Rightarrow T(n) \in ?$

$T(n) = 4T(n/2) + n^2 \Rightarrow T(n) \in ?$

$T(n) = 4T(n/2) + n^3 \Rightarrow T(n) \in ?$

Master theorem Examples (cont.)

Solutions: $T(n) = 4T(n/2) + n \Rightarrow T(n) \in \Theta(n^2)$

$T(n) = 4T(n/2) + n^2 \Rightarrow T(n) \in \Theta(n^2 \log n)$

$T(n) = 4T(n/2) + n^3 \Rightarrow T(n) \in \Theta(n^3)$

Example: Divide-and-Conquer Analysis

- For example, the recurrence for the number of additions A(n) made by the divide-and-conquer sum-computation algorithm on inputs of size $n = 2^k$:

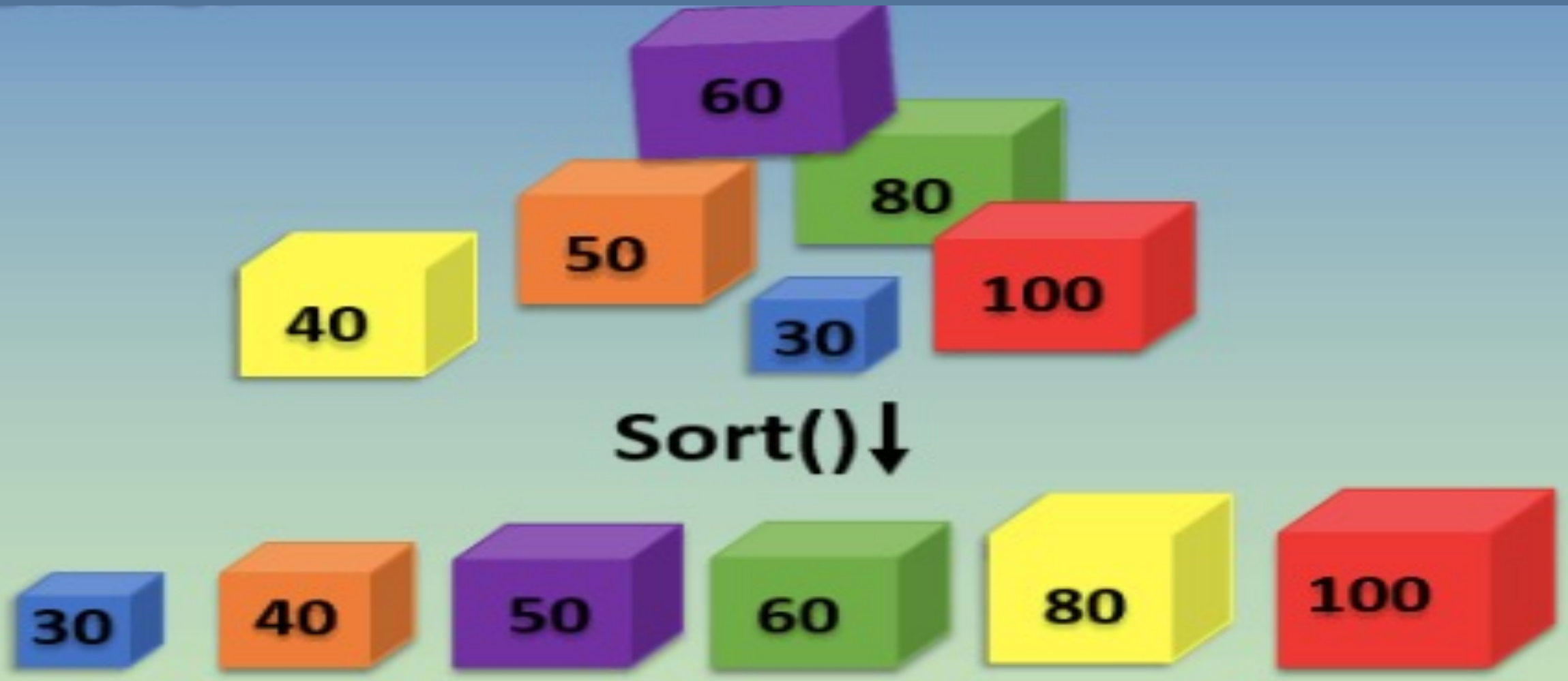
$$A(n) = 2A(n/2) + 1$$

- For this example, a = 2, b = 2, and d = 0; hence, since $a > b^d$,

$$A(n) \in \Theta(n^{\log_b a}) = \Theta(n^{\log_2 2}) = \Theta(n).$$

- We were able to find the solution's efficiency class without going through the drudgery of solving the recurrence.

Merge Sort



Merge Sort

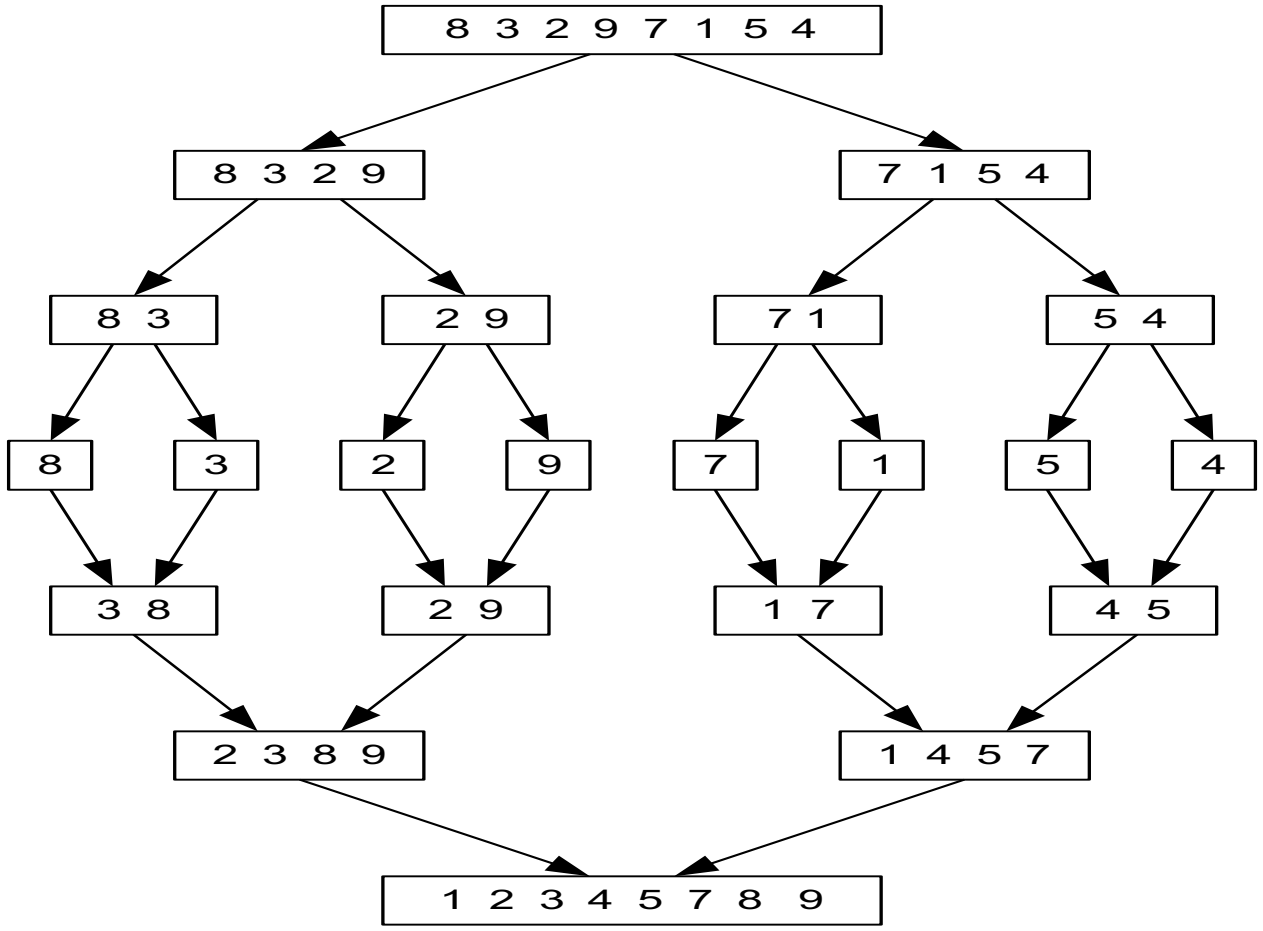
Merge Sort Steps:

- Split array A[0..*n*-1] in two about equal halves and make copies of each half in arrays B and C
- Sort arrays B and C recursively
- Merge sorted arrays B and C into array A as follows:
  - Repeat the following until no elements remain in one of the arrays:
    - compare the first elements in the remaining unprocessed portions of the arrays
    - copy the smaller of the two into A, while incrementing the index indicating the unprocessed portion of that array
  - Once all elements in one of the arrays are processed, copy the remaining unprocessed elements from the other array into A.

Merge Sort

6 5 3 1 8 7 2 4

Merge Sort Example



Merge Sort Example

- Apply merge sort algorithm to sort the following array.

7 5 8 2 3 9 11 10

Pseudocode of Merge Sort

ALGORITHM *Mergesort*$(A[0..n-1])$

```
//Sorts array A[0..n − 1] by recursive mergesort
//Input: An array A[0..n − 1] of orderable elements
//Output: Array A[0..n − 1] sorted in nondecreasing order
if n > 1
    copy A[0..⌊n/2⌋ − 1] to B[0..⌊n/2⌋ − 1]
    copy A[⌊n/2⌋..n − 1] to C[0..⌈n/2⌉ − 1]
    Mergesort(B[0..⌊n/2⌋ − 1])
    Mergesort(C[0..⌈n/2⌉ − 1])
    Merge(B, C, A)
```

Pseudocode of Merge

ALGORITHM *Merge*$(B[0..p-1], C[0..q-1], A[0..p+q-1])$

//Merges two sorted arrays into one sorted array
//Input: Arrays $B[0..p-1]$ and $C[0..q-1]$ both sorted
//Output: Sorted array $A[0..p+q-1]$ of the elements of B and C
$i \leftarrow 0;\ j \leftarrow 0;\ k \leftarrow 0$
while $i < p$ **and** $j < q$ **do**
    if $B[i] \leq C[j]$
        $A[k] \leftarrow B[i];\ i \leftarrow i+1$
    else $A[k] \leftarrow C[j];\ j \leftarrow j+1$
    $k \leftarrow k+1$
if $i = p$
    copy $C[j..q-1]$ to $A[k..p+q-1]$
else copy $B[i..p-1]$ to $A[k..p+q-1]$

Analysis of Merge Sort

- Assuming for simplicity that n is a power of 2, the recurrence relation for the number of key comparisons C(n) is

$$C(n) = 2C(n/2) + C_{merge}(n) \quad \text{for } n > 1, \quad C(1) = 0.$$

- Let us analyze $C_{merge}(n)$, the number of key comparisons performed during the merging stage.
- At each step, exactly one comparison is made, after which the total number of elements in the two arrays still needing to be processed is reduced by 1.

Analysis of Merge Sort

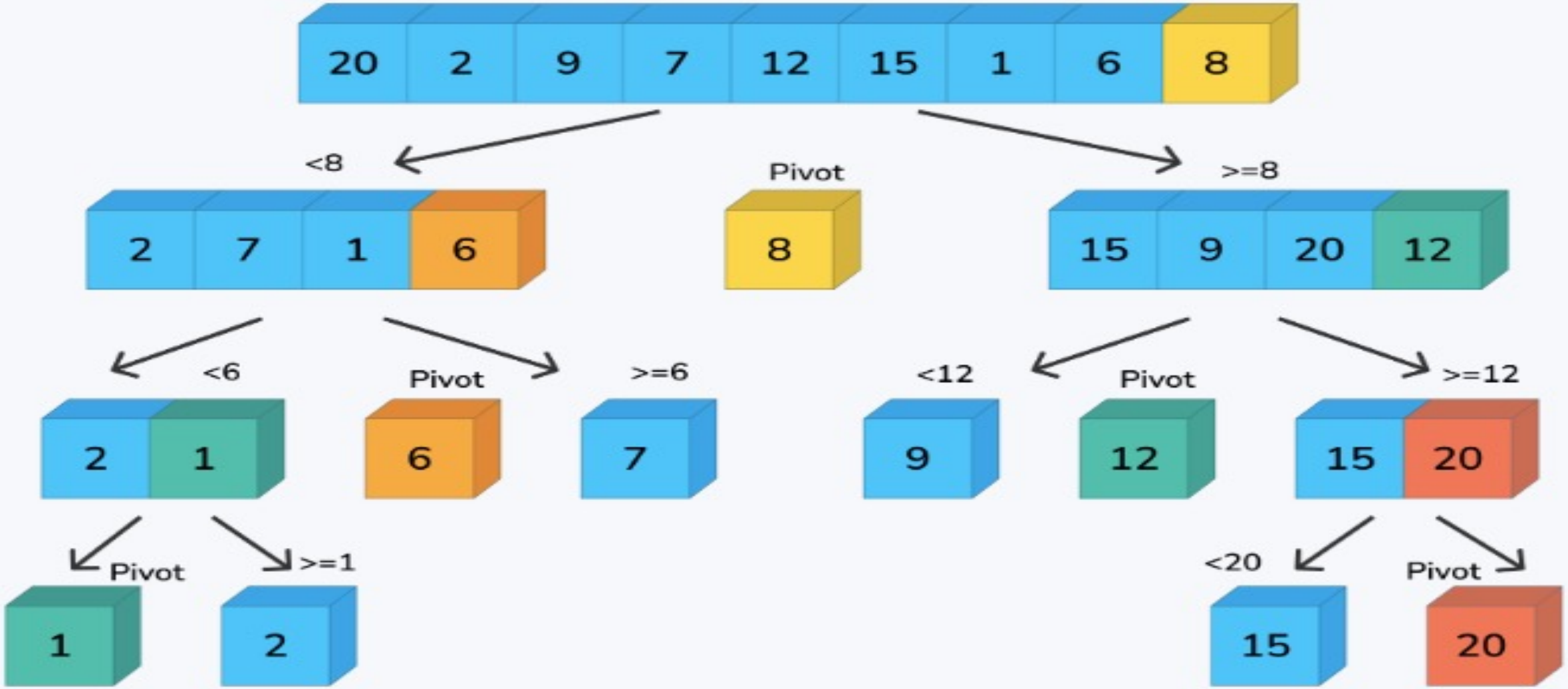
- In the worst case, neither of the two arrays becomes empty before the other one contains just one element (e.g., smaller elements may come from the alternating arrays).
- Therefore, for the worst case, $C_{merge}(n) = n - 1$, and we have the recurrence

$$C_{worst}(n) = 2C_{worst}(n/2) + n - 1 \quad \text{for } n > 1, \quad C_{worst}(1) = 0.$$

- According to the Master Theorem, $C_{worst}(n) \in \Theta(n \log n)$.
- It is easy to find the exact solution to the worst-case recurrence using backward substitution:

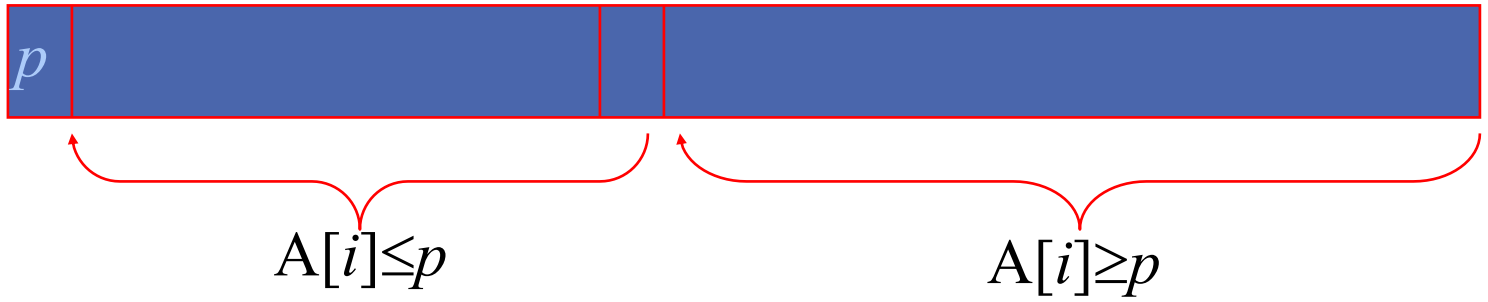
$$C_{worst}(n) = n \log_2 n - n + 1.$$

Quick Sort


20 2 9 7 12 15 1 6 8
<8
Pivot
>=8
2 7 1 6
8
15 9 20 12
<6
Pivot
>=6
<12
Pivot
>=12
2 1
6
7
9
12
15 20
Pivot
>=1
<20
Pivot
1
2
15
20

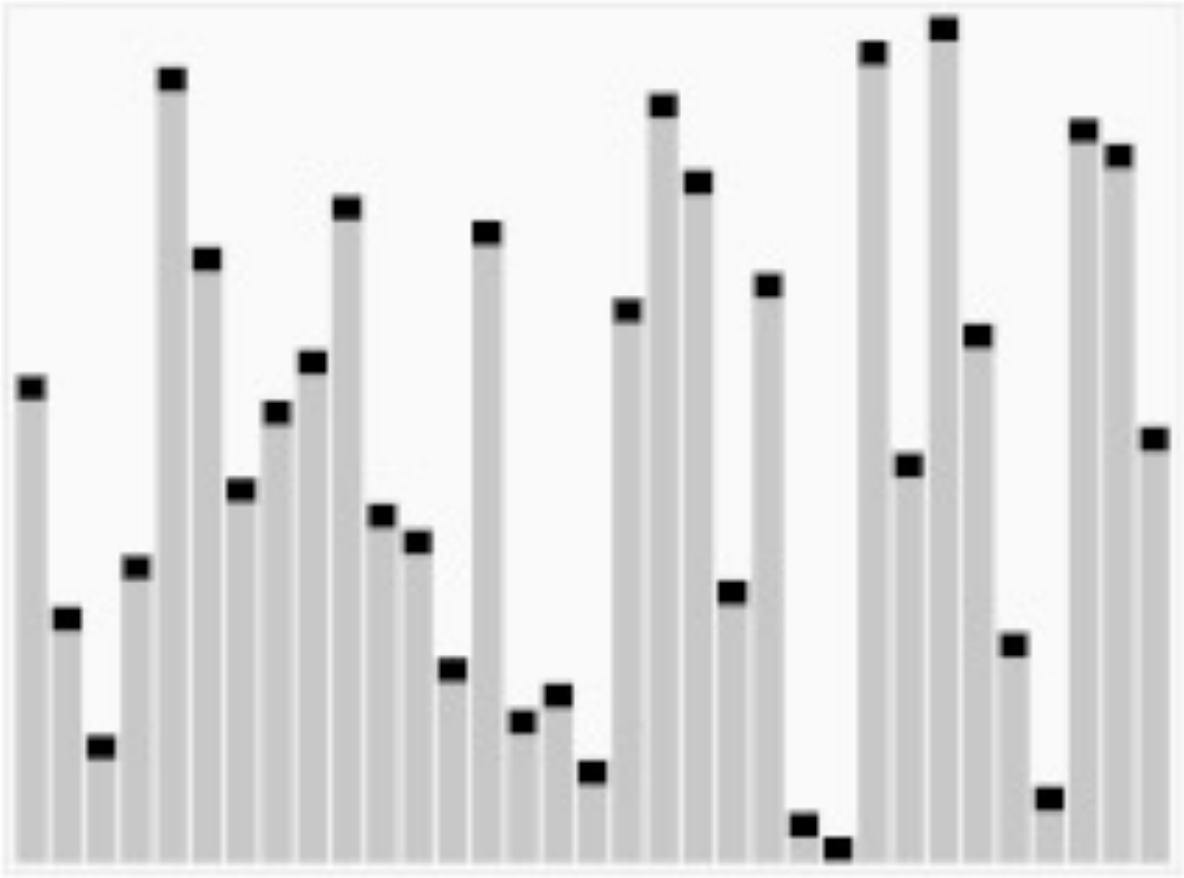
Quick Sort

- Select a *pivot* (partitioning element) – here, the first element
- Rearrange the list so that all the elements in the first *s* positions are smaller than or equal to the pivot and all the elements in the remaining *n-s* positions are larger than or equal to the pivot.



- Exchange the pivot with the last element in the first (i.e., ≤) subarray — the pivot is now in its final position
- Sort the two subarrays recursively

Quick Sort



Quicksort-Partitioning

- Unlike mergesort, which divides its input elements according to their position in the array, quicksort divides them according to their value.
- We already encountered this idea of an array partition in the selection problem.
- A partition is an arrangement of the array's elements so that all the elements to the left of some element A[s] are less than or equal to A[s], and all the elements to the right of A[s] are greater than or equal to it:

$$\underbrace{A[0] \ldots A[s-1]}_{\text{all are} \le A[s]} \quad A[s] \quad \underbrace{A[s+1] \ldots A[n-1]}_{\text{all are} \ge A[s]}$$

Quick Sort-Partitioning

- After a partition is achieved, A[s] will be in its final position in the sorted array, and we can continue sorting the two subarrays to the left and to the right of A[s] independently (e.g., by the same method).
- Difference with merge Sort:
  - **Merge Sort**: The division of the problem into two subproblems is immediate and the entire work happens in combining their solutions;
  - **Quick Sort**: The entire work happens in the division stage, with no work required to combine the solutions to the subproblems.

Quicksort

ALGORITHM *Quicksort*$(A[l..r])$

//Sorts a subarray by quicksort
//Input: Subarray of array $A[0..n-1]$, defined by its left and right
// indices l and r
//Output: Subarray $A[l..r]$ sorted in nondecreasing order
if $l < r$
  $s \leftarrow$ *Partition*$(A[l..r])$ //s is a split position
  Quicksort$(A[l..s-1])$
  Quicksort$(A[s+1..r])$

Hoare's Partitioning Algorithm

- Scan the subarray from both ends, comparing the subarray's elements to the pivot.
- The left-to-right scan, denoted by index pointer *i*, starts with the second element.
- Since we want elements smaller than the pivot to be in the left part of the subarray, this scan skips over elements that are smaller than the pivot and stops upon encountering the first element greater than or equal to the pivot.

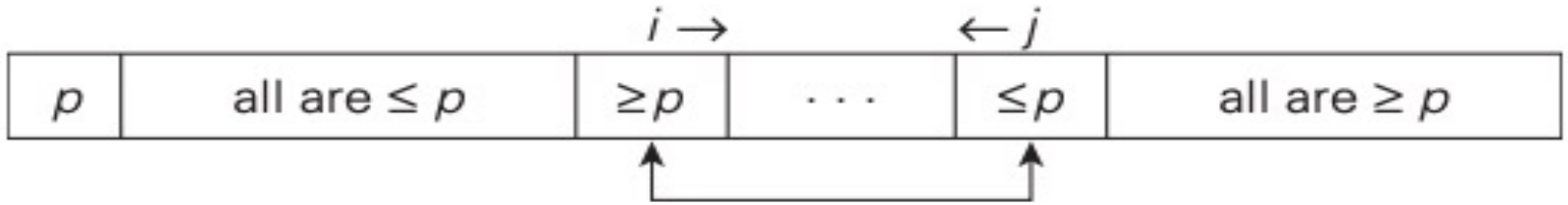
Hoare's Partitioning Algorithm

- The right-to-left scan, denoted by index pointer *j*, starts with the last element of the subarray.
- Since we want elements larger than the pivot to be in the right part of the subarray, this scan skips over elements that are larger than the pivot and stops on encountering the first element smaller than or equal to the pivot.
- For example, if we did otherwise for an array of n equal elements, we would have gotten a split into subarrays of sizes n − 1 and 0, reducing the problem size just by 1 after scanning the entire array.

Hoare's Partitioning Algorithm

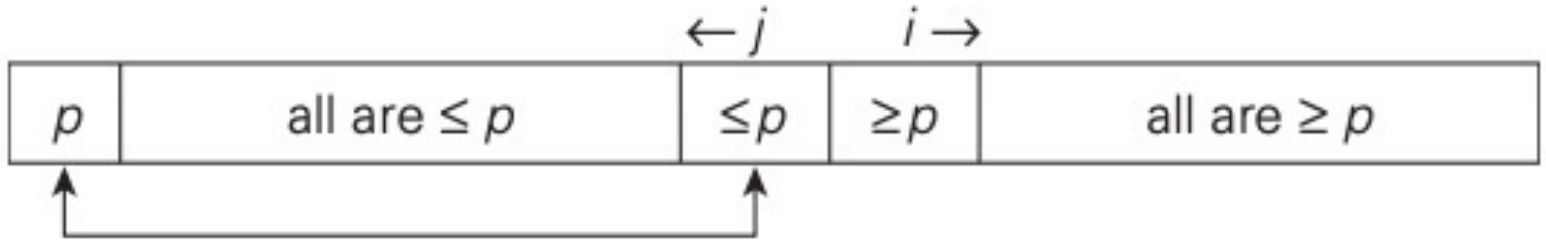
- After both scans stop, three situations may arise, depending on whether or not the scanning indices have crossed.

1- If scanning indices i and j have not crossed, i.e., i < j, we simply exchange A[i] and A[j] and resume the scans by incrementing i and decrementing j, respectively:



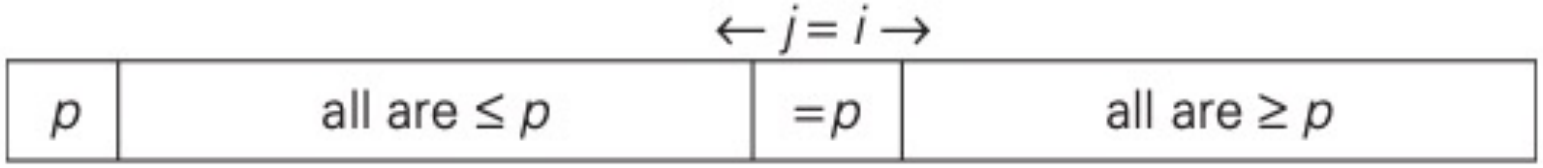
Hoare's Partitioning Algorithm

2- If the scanning indices have crossed over, i.e., i > j, we will have partitioned the subarray after exchanging the pivot with A[j]:



Hoare's Partitioning Algorithm

3- Finally, if the scanning indices stop while pointing to the same element, i.e., i = j, the value they are pointing to must be equal to p. Thus, we have the subarray partitioned, with the split position s = i = j



Hoare's Partitioning Algorithm

ALGORITHM *HoarePartition*$(A[l..r])$

//Partitions a subarray by Hoare's algorithm, using the first element
// as a pivot
//Input: Subarray of array $A[0..n-1]$, defined by its left and right
// indices l and r $(l < r)$
//Output: Partition of $A[l..r]$, with the split position returned as
// this function's value
$p \leftarrow A[l]$
$i \leftarrow l;\ j \leftarrow r+1$
repeat
  repeat $i \leftarrow i+1$ **until** $A[i] \geq p$
  repeat $j \leftarrow j-1$ **until** $A[j] \leq p$
  swap$(A[i], A[j])$
until $i \geq j$
swap$(A[i], A[j])$ //undo last swap when $i \geq j$
swap$(A[l], A[j])$
return j

Quick Sort Example

- Apply quick sort algorithm to sort the following array.

5 3 1 9 8 2 4 7

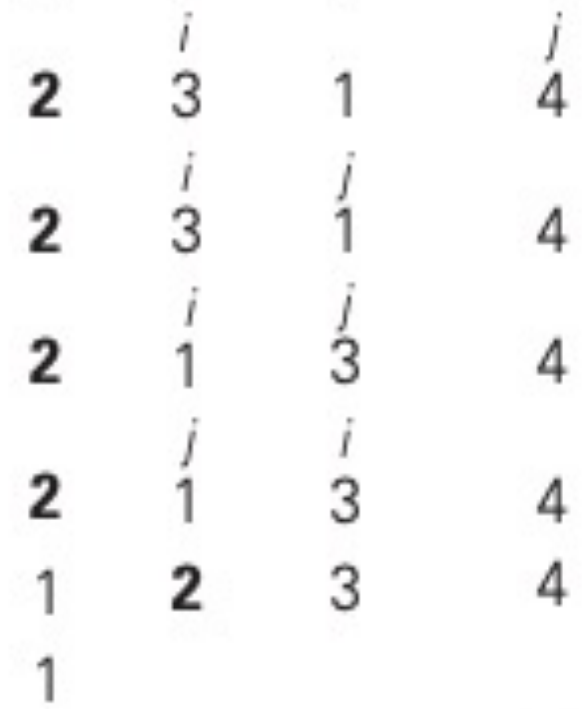
Quick Sort Example

Step 1:

0	1	2	3	4	5	6	7
	i						*j*
5	3	1	9	8	2	4	7
			i			*j*	
5	3	1	9	8	2	4	7
			i			*j*	
5	3	1	4	8	2	9	7
				i	*j*		
5	3	1	4	8	2	9	7
				i	*j*		
5	3	1	4	2	8	9	7
				j	*i*		
5	3	1	4	2	8	9	7
2	3	1	4	**5**	8	9	7

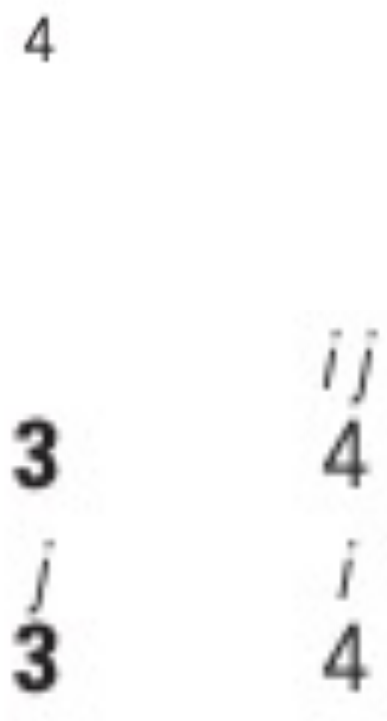
Quick Sort Example

Step 2: 2 3 1 4 **5** 8 9 7



Quick Sort Example

Step 3:



Quick Sort Example

Step 4:

Quick Sort Example

0	1	2	3	4	5	6	7
5	3	1	9	8	2	4	7
5	3	1	9	8	2	4	7
5	3	1	4	8	2	9	7
5	3	1	4	8	2	9	7
5	3	1	4	2	8	9	7
5	3	1	4	2	8	9	7
2	3	1	4	**5**	8	9	7
2	3	1	4				
2	3	1	4				
2	1	3	4				
2	1	3	4				
1	**2**	3	4				
1							
		3	4				
		3	4				
			4				
					8	9	7
					8	7	9
					8	7	9
					7	**8**	9
					7		
							9

Result: 1 2 3 4 5 7 8 9

Quick Sort Example

- Apply quick sort algorithm to sort the following array.

7 5 8 2 3 9 11 10 4

Analysis of Quick Sort

- Best case: Split in the middle — $\Theta(n \log n)$
- Worst case: Sorted array! — $\Theta(n^2)$
- Average case: Random arrays — $\Theta(n \log n)$
- Improvements:
  - Better pivot selection methods such as ***randomized quicksort*** that uses a random element or the ***median-of-three*** method that uses the median of the leftmost, rightmost, and the middle element of the array
  - Switching to insertion sort on very small subarrays (between 5 and 15 elements for most computer systems) or not sorting small subarrays at all and finishing the algorithm with insertion sort applied to the entire nearly sorted array
  - These combine to 20-25% improvement
- Considered the method of choice for internal sorting of large files ($n \geq 10{,}000$)

Georgia State University

CSC 4520/6520
Design & Analysis of Algorithms

CHAPTER 5: DIVIDE-AND-CONQUER

Abdullah Bal, PhD
Spring 2024

Multiplication of Large Integers

- Some applications, notably modern cryptography, require manipulation of integers that are over 100 decimal digits long.
- Since such integers are too long to fit in a single word of a modern computer, they require special treatment.
- if we use the conventional pen-and-pencil algorithm for multiplying two n-digit integers, each of the n digits of the first number is multiplied by each of the n digits of the second number for the total of n^2 digit multiplications.
- Though it might appear that it would be impossible to design an algorithm with fewer than n^2 digit multiplications, this turns out not to be the case.

Multiplication of Large Integers

- Consider the problem of multiplying two (large) *n*-digit integers represented by arrays of their digits such as:

  A = 12345678901357986429 B = 87654321284820912836

  The grade-school algorithm:

$$
\begin{array}{r}
a_1\ a_2 \ldots\ a_n \\
b_1\ b_2 \ldots\ b_n \\
\hline
(d_{10})\ d_{11} d_{12} \ldots d_{1n} \\
(d_{20})\ d_{21} d_{22} \ldots d_{2n} \\
\ldots\ \ldots\ \ldots\ \ldots\ \ldots\ \ldots\ \ldots \\
\hline
(d_{n0})\ d_{n1} d_{n2} \ldots d_{nn}
\end{array}
$$

- Efficiency: n^2 one-digit multiplications

First Step: Divide-and-Conquer Algorithm

- A small example: $A * B$ where A = 2135 and B = 4014

$$A = (21 \cdot 10^2 + 35),$$

$$B = (40 \cdot 10^2 + 14)$$

So,

$$A * B = (21 \cdot 10^2 + 35) * (40 \cdot 10^2 + 14)$$

$$= 21 * 40 \cdot 10^4 + (21 * 14 + 35 * 40) \cdot 10^2 + 35 * 14$$

In general, if $A = A_1A_2$ and $B = B_1B_2$ (where A and B are *n*-digit, A_1, A_2, B_1, B_2 are *n/2*-digit numbers),

$$A * B = A_1 * B_1 \cdot 10^n + (A_1 * B_2 + A_2 * B_1) \cdot 10^{n/2} + A_2 * B_2.$$ (4 Multiplications and 3 add)

First Step: Divide-and-Conquer Algorithm

$A * B = A_1 * B_1 \cdot 10^n + (A_1 * B_2 + A_2 * B_1) \cdot 10^{n/2} + A_2 * B_2$ (4 Multiplications and 3 add)

- Recurrence for the number of one-digit multiplications M(n):

$$M(n) = 4M(n/2), \quad M(1) = 1$$

- Solution: $M(n) = n^2$

Second Step: Divide-and-Conquer Algorithm

- The idea is to decrease the number of multiplications from 4 to 3:

$$A * B = A_1 * B_1 \cdot 10^n + (A_1 * B_2 + A_2 * B_1) \cdot 10^{n/2} + A_2 * B_2 \quad \text{(4 Multiplications and 3 add)}$$
$$= c_2 \cdot 10^n + c_1 \cdot 10^{n/2} + c_0$$

$c_2 = A_1 * B_1$

$c_0 = A_2 * B_2$

$c_1 = A_1 * B_2 + A_2 * B_1$

- Rewrite c_1

$(A_1 + A_2) * (B_1 + B_2) = A_1 * B_1 + (A_1 * B_2 + A_2 * B_1) + A_2 * B_2$,

i.e., $(A_1 * B_2 + A_2 * B_1) = (A_1 + A_2) * (B_1 + B_2) - A_1 * B_1 - A_2 * B_2$,

$c_1 = (A_1 + A_2) * (B_1 + B_2) - (c_2 + c_0)$

- Requires only 3 multiplications at the expense of extra add/sub.

Second Step: Divide-and-Conquer Algorithm

- Recurrence for the number of multiplications M(n):

$$M(n) = 3M(n/2), \quad M(1) = 1$$

- Solution: Backward substation $M(n) = 3^{\log_2 n} = n^{\log_2 3} \approx n^{1.585}$

Master Theorem
If $a < b^d$, $T(n) \in \Theta(n^d)$
If $a = b^d$, $T(n) \in \Theta(n^d \log n)$
If $a > b^d$, $T(n) \in \Theta(n^{\log_b a})$

($a^{\log_b c} = c^{\log_b a}$)

Strassen's Matrix Multiplication

- Strassen observed [1969] that the product of two matrices can be computed as follows:

$$\begin{pmatrix} C_{00} & C_{01} \\ C_{10} & C_{11} \end{pmatrix} = \begin{pmatrix} A_{00} & A_{01} \\ A_{10} & A_{11} \end{pmatrix} * \begin{pmatrix} B_{00} & B_{01} \\ B_{10} & B_{11} \end{pmatrix}$$

$$C_{00}=A_{00}*B_{00} + A_{01}*B_{10}$$
$$C_{01}=A_{00}*B_{01} + A_{01}*B_{11}$$
$$C_{10}=A_{10}*B_{00} + A_{11}*B_{10}$$
$$C_{11}=A_{10}*B_{01} + A_{11}*B_{11}$$

Formulas for Strassen's Algorithm

$M_1 = (A_{00} + A_{11}) * (B_{00} + B_{11})$

$M_2 = (A_{10} + A_{11}) * B_{00}$

$M_3 = A_{00} * (B_{01} - B_{11})$

$M_4 = A_{11} * (B_{10} - B_{00})$

$M_5 = (A_{00} + A_{01}) * B_{11}$

$M_6 = (A_{10} - A_{00}) * (B_{00} + B_{01})$

$M_7 = (A_{01} - A_{11}) * (B_{10} + B_{11})$

Strassen's Matrix Multiplication

- Strassen presented the product of two matrices using Strassen formulas:

$$\begin{pmatrix} C_{00} & C_{01} \\ C_{10} & C_{11} \end{pmatrix} = \begin{pmatrix} A_{00} & A_{01} \\ A_{10} & A_{11} \end{pmatrix} * \begin{pmatrix} B_{00} & B_{01} \\ B_{10} & B_{11} \end{pmatrix}$$

$$C_{00}=A_{00}*B_{00} + A_{01}*B_{10}$$
$$C_{01}=A_{00}*B_{01} + A_{01}*B_{11}$$
$$C_{10}=A_{10}*B_{00} + A_{11}*B_{10}$$
$$C_{11}=A_{10}*B_{01} + A_{11}*B_{11}$$

$$= \begin{pmatrix} M_1 + M_4 - M_5 + M_7 & M_3 + M_5 \\ M_2 + M_4 & M_1 + M_3 - M_2 + M_6 \end{pmatrix}$$

Analysis of Strassen's Algorithm

- If n is not a power of 2, matrices can be padded with zeros.
- Number of multiplications:

$$M(n) = 7M(n/2), \quad M(1) = 1$$

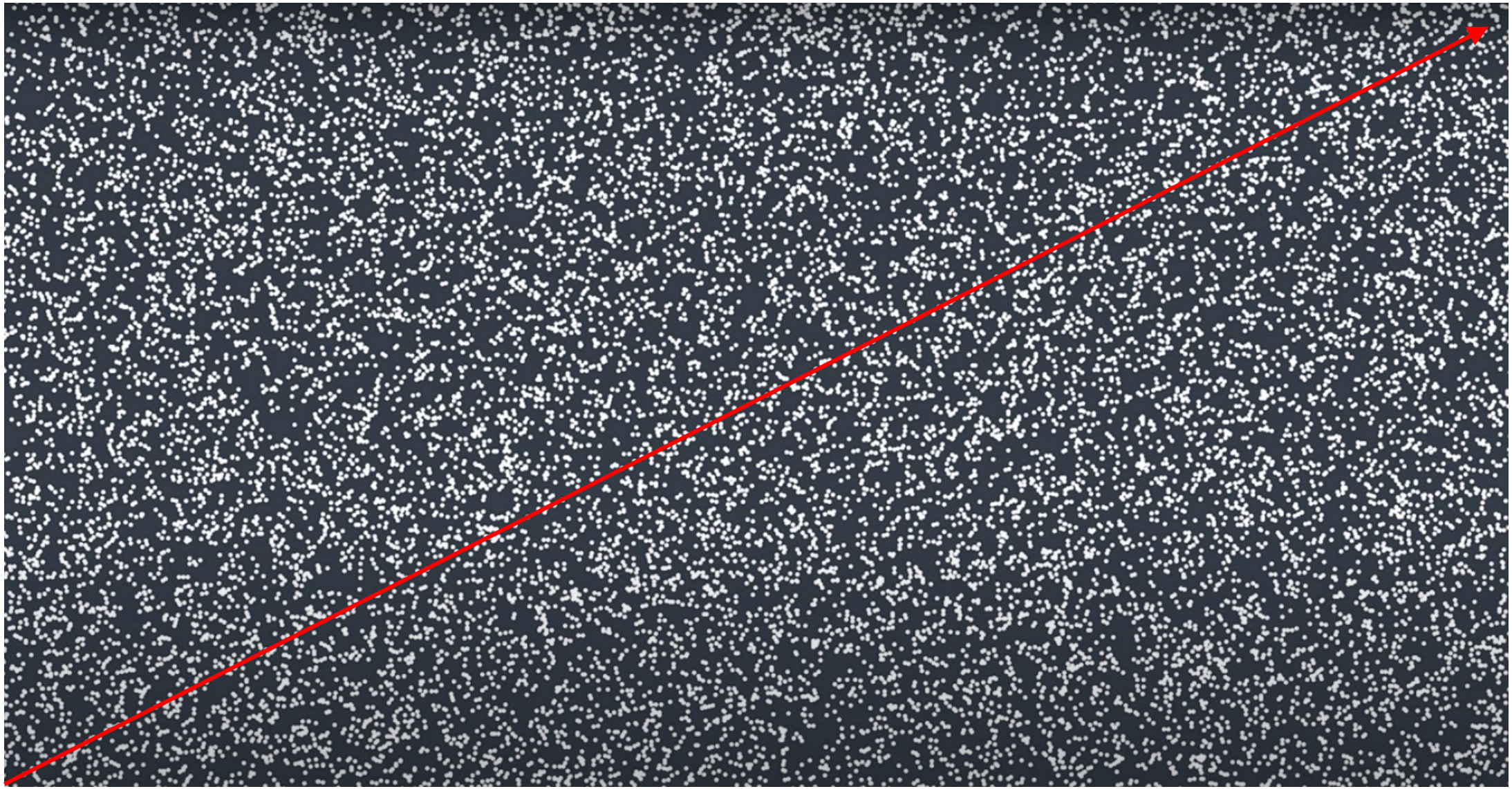
- Solution: $M(n) = 7^{\log_2 n} = n^{\log_2 7} \approx n^{2.807}$ vs. n^3 of brute-force algorithm
- Algorithms with better asymptotic efficiency are known but they are even more complex.

Closest-Pair Problem by Divide-and-Conquer

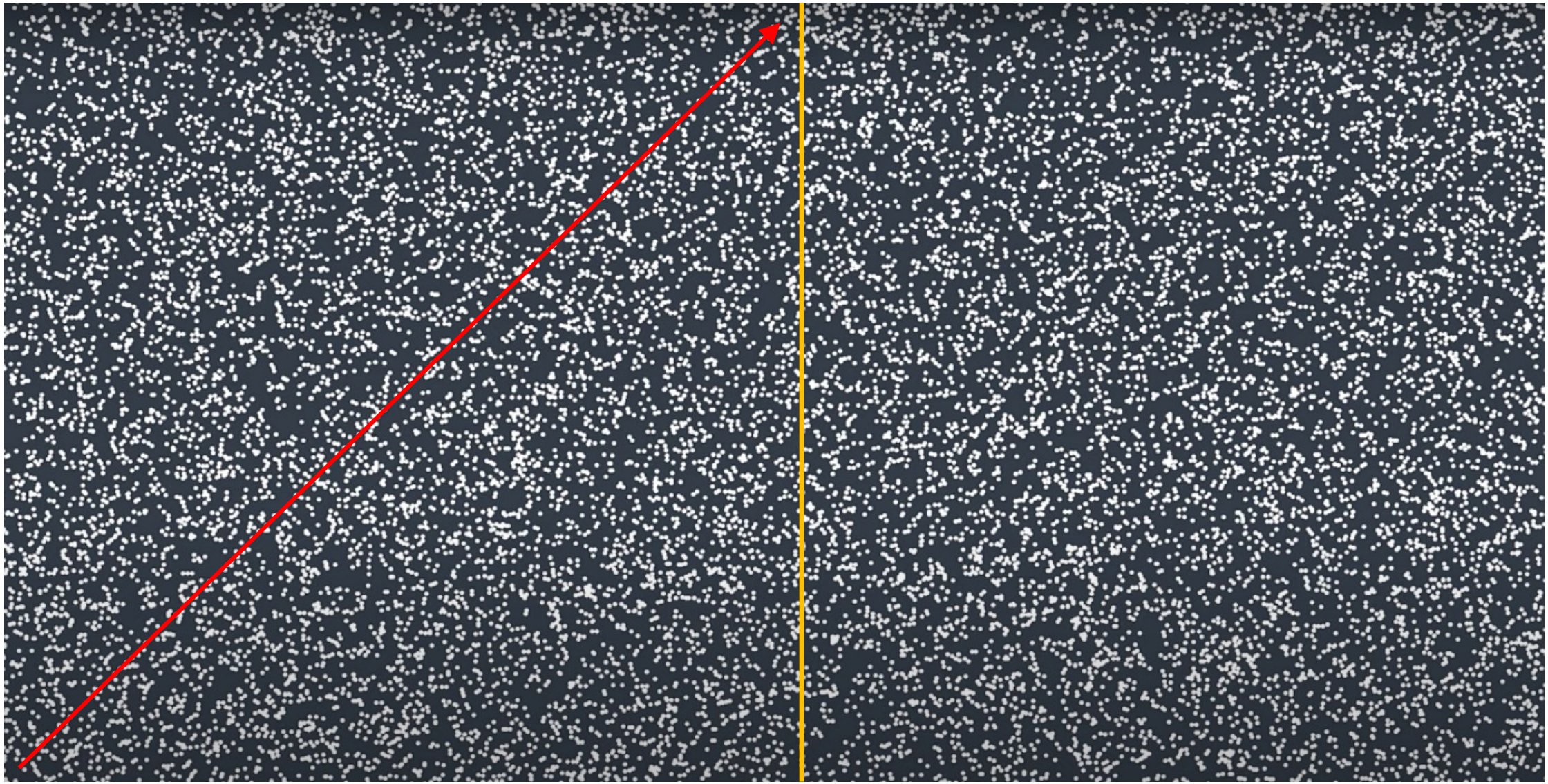
Closest-Pair Problem by Divide-and-Conquer

Closest-Pair Problem by Divide-and-Conquer

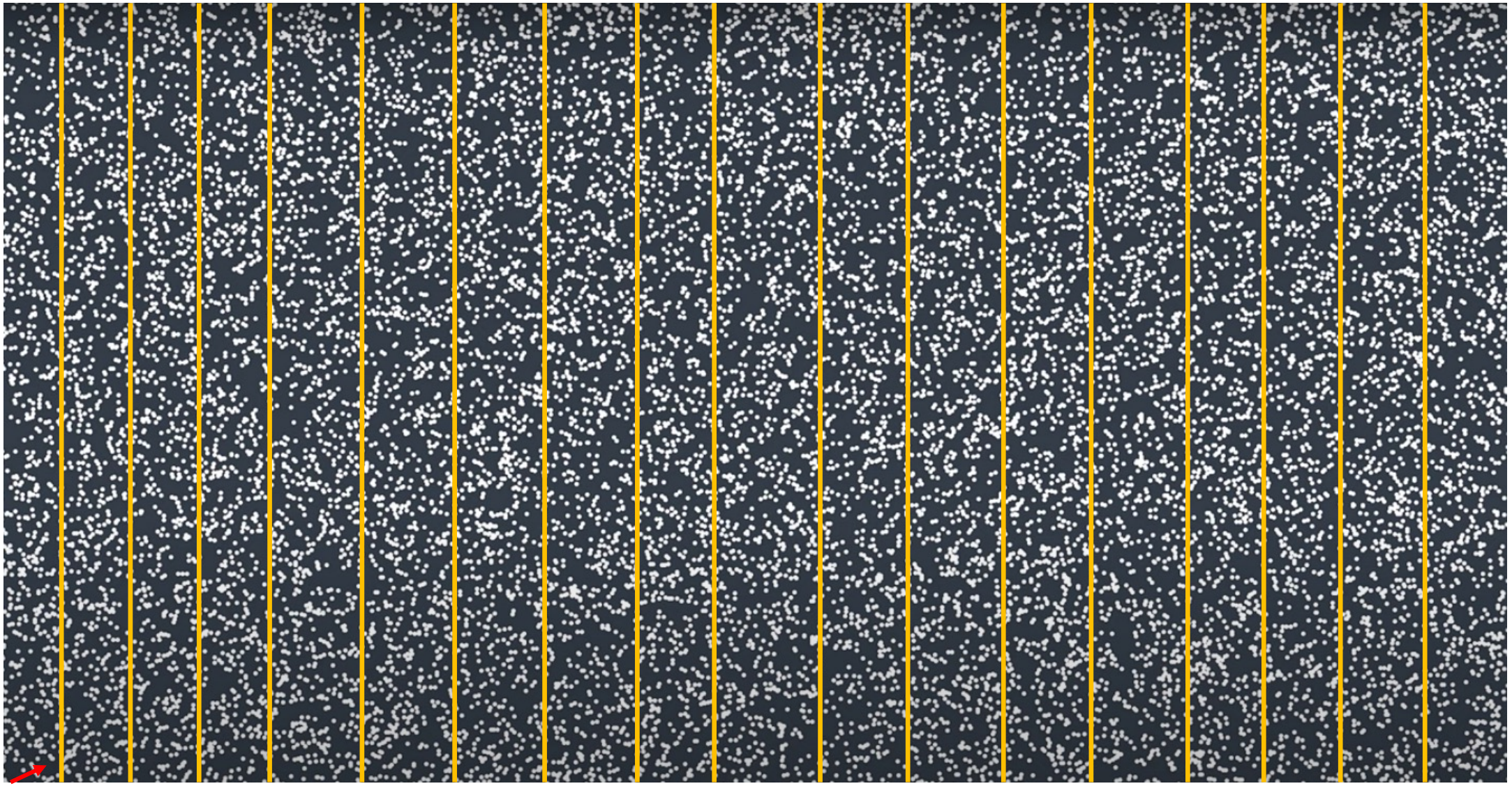
Closest-Pair Problem by Divide-and-Conquer



Closest-Pair Problem by Divide-and-Conquer

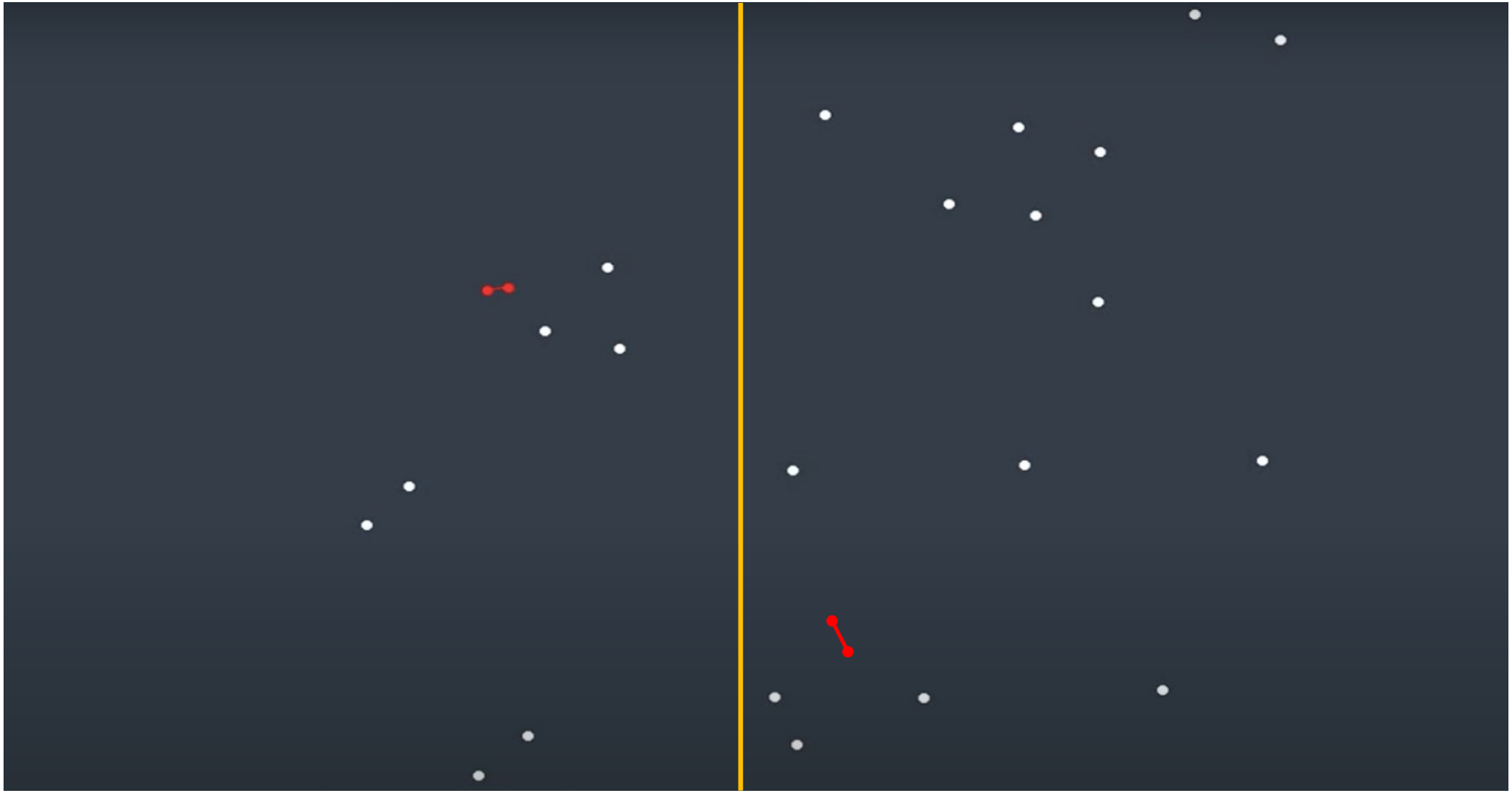


Closest-Pair Problem by Divide-and-Conquer

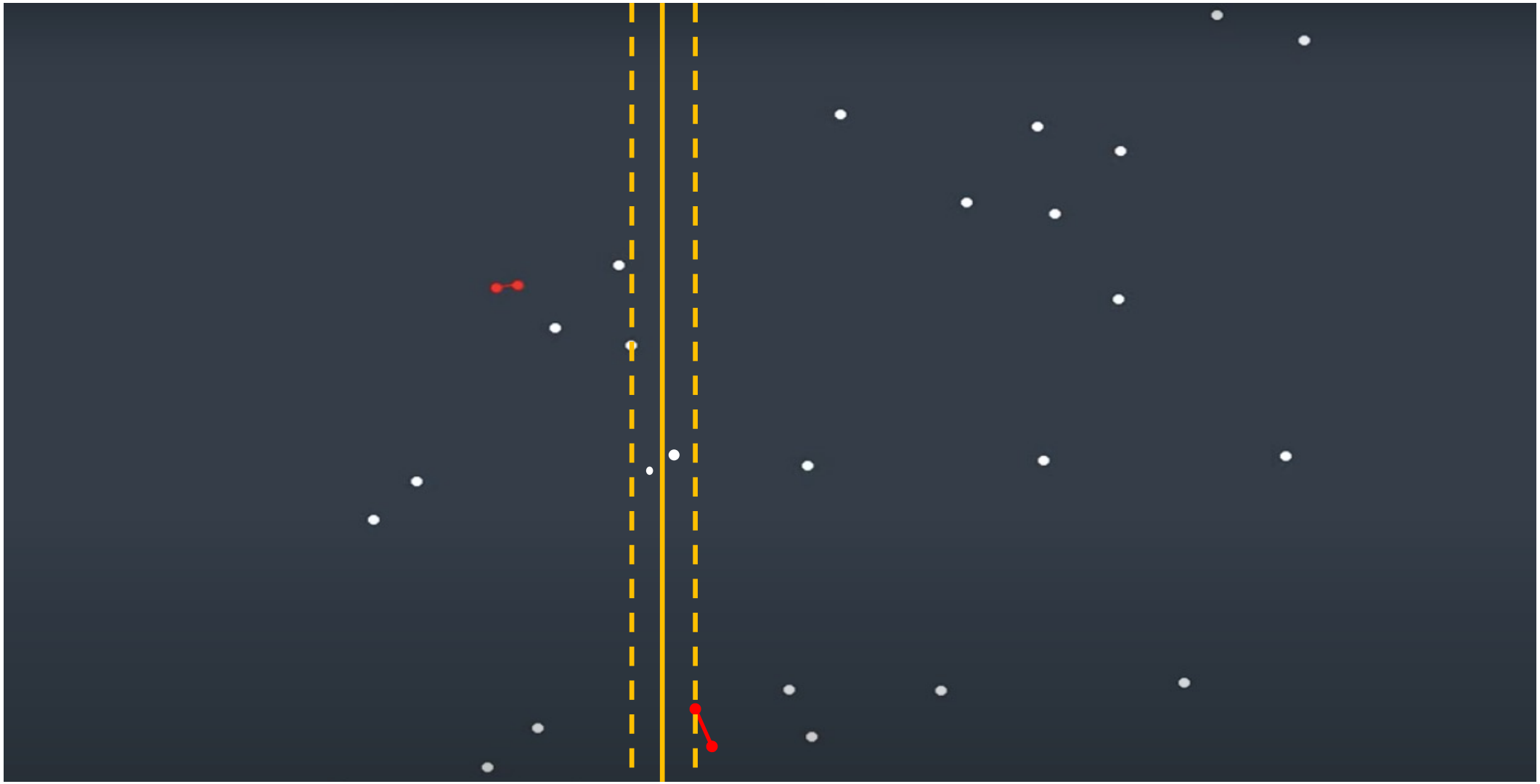


Closest-Pair Problem by Divide-and-Conquer

Closest-Pair Problem by Divide-and-Conquer

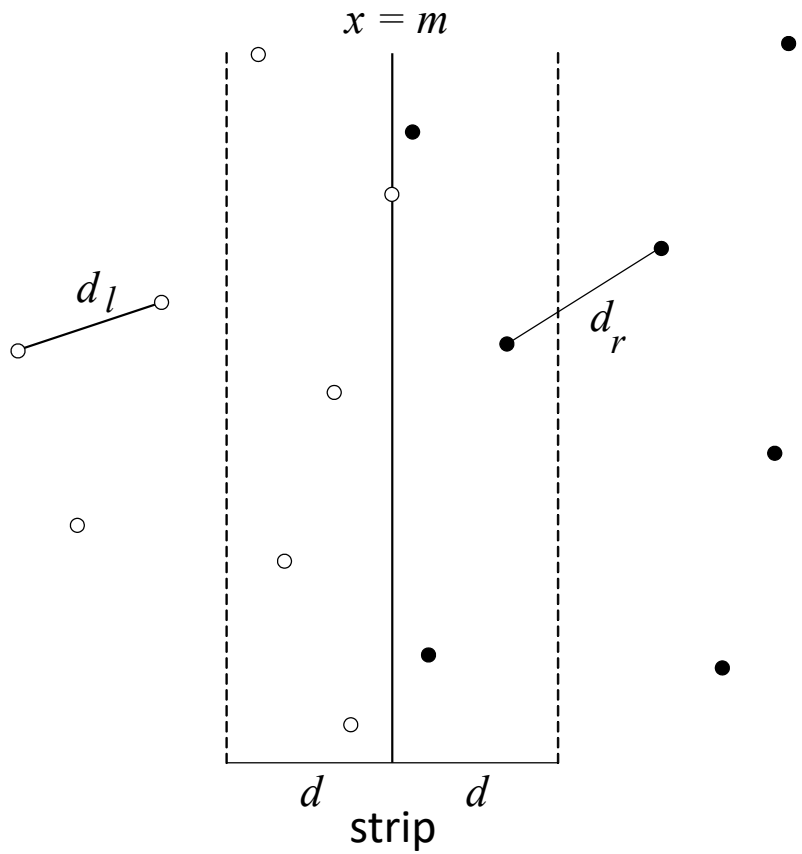


Closest-Pair Problem by Divide-and-Conquer



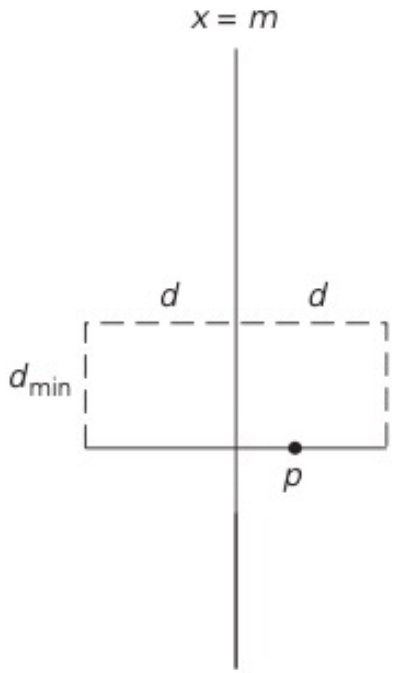
Closest-Pair Problem by Divide-and-Conquer

Step 1: Divide the points given into two subsets P_l and P_r by a vertical line $x = m$ so that half the points lie to the left or on the line and half the points lie to the right or on the line.



Closest-Pair Problem by Divide-and-Conquer

- Rectangle that may contain points closer than d_{min} to point p.



strip

Closest Pair by Divide-and-Conquer (cont.)

Step 2: Find recursively the closest pairs for the left and right subsets.

Step 3: Set $d = \min\{d_l, d_r\}$

- We can limit our attention to the points in the symmetric vertical strip S of width 2d as possible closest pair.
- The points are stored and processed in increasing order of their y coordinates.

Step 4: Scan the points in the vertical strip *S* from the lowest up.

- For every point $p(x,y)$ in the strip, inspect points in the strip that may be closer to p than d.

Efficiency of the Closest-Pair Algorithm

- The algorithm spends linear time both for dividing the problem into two problems half the size and combining the obtained solutions.
- Running time of the algorithm is described by

$$T(n) = 2T(n/2) + f(n), \text{ where } f(n) \in \Theta(n) \text{ (Finding median)}$$

- By the Master Theorem (with $a = 2$, $b = 2$, $d = 1$)

$$T(n) \in \Theta(n \log n).$$

Master Theorem
If $a < b^d$, $T(n) \in \Theta(n^d)$
If $a = b^d$, $T(n) \in \Theta(n^d \log n)$
If $a > b^d$, $T(n) \in \Theta(n^{\log_b a})$

Georgia State University

CSC 4520/6520
Design & Analysis of Algorithms

CHAPTER 6: TRANSFORM-AND-CONQUER (INSTANCE SIMPLIFICATION)

Abdullah Bal, PhD
Spring 2024

Transform and Conquer

- Instance simplification: This group of techniques solves a problem by a transformation to a simpler/more convenient instance of the same problem.
- Representation change: A different representation of the same instance
- Problem reduction: A different problem for which an algorithm is already available

Instance simplification

Presorting

Presorting

- Solve a problem's instance by transforming it into another simpler/easier instance of the same problem

Presorting

- Many problems involving lists are easier when list is sorted, e.g.
- Searching
- Computing the median (selection problem)
- Checking if all elements are distinct (element uniqueness)

Also:

- Topological sorting helps solving some problems for dags.
- Presorting is used in many geometric algorithms.

Example 1: Element Uniqueness with presorting

Bruteforce

- Checking element uniqueness in an array
- The bruteforce algorithm compared pairs of the array's elements until either two equal elements were found or no more pairs were left.
- Its worst-case efficiency was in $\Theta(n^2)$.

Example 1: Element Uniqueness with presorting

Presorting

- Alternatively, we can sort the array first and then check only its consecutive elements:
  - if the array has equal elements, a pair of them must be next to each other, and vice versa.

Example 1: Element Uniqueness with presorting

ALGORITHM *PresortElementUniqueness*$(A[0..n-1])$

//Solves the element uniqueness problem by sorting the array first
//Input: An array $A[0..n-1]$ of orderable elements
//Output: Returns "true" if A has no equal elements, "false" otherwise
sort the array A
for $i \leftarrow 0$ **to** $n-2$ **do**
    if $A[i] = A[i+1]$ **return false**
return true

Example 1: Element Uniqueness with presorting

- The running time of this algorithm is the sum of the time spent on sorting and the time spent on checking consecutive elements.

$$T(n) = T_{sort}(n) + T_{scan}(n)$$

- Since the former (sorting) requires at least n log n comparisons and the latter (scanning) needs no more than n − 1 comparisons, it is the sorting part that will determine the overall efficiency of the algorithm.

Example 1: Element Uniqueness with presorting

- So, if we use a quadratic sorting algorithm here, the entire algorithm will not be more efficient than the brute-force one.
- But if we use a good sorting algorithm, such as merge sort, with worst-case efficiency in Θ(n log n), the worst-case efficiency of the entire presorting-based algorithm will be also in Θ(n log n):

Example 1: Element Uniqueness with presorting

- Presorting-based algorithm
  - Stage 1: sort by efficient sorting algorithm (e.g. merge sort)
  - Stage 2: scan array to check pairs of adjacent elements
  - Efficiency: $\Theta(n \log n) + O(n) = \Theta(n \log n)$

- Brute force algorithm
  - Compare all pairs of elements
  - Efficiency: $O(n^2)$

Example 2: Computing a Mode

- A ***mode*** is a value that occurs most often in a given list of numbers.
- For example, for 5, 1, 5, 7, 6, 5, 7, the mode is 5.
- If several different values occur most often, any of them can be considered a mode.

Example 2: Computing a Mode

Bruteforce

- The brute-force approach to computing a mode would scan the list and compute the frequencies of all its distinct values, then find the value with the largest frequency.
- In order to implement this idea, we can store the values already encountered, along with their frequencies, in a separate list.
- On each iteration, the i^{th} element of the original list is compared with the values already encountered by traversing this auxiliary list.
- If a matching value is found, its frequency is incremented; otherwise, the current element is added to the list of distinct values seen so far with a frequency of 1.

Example 2: Computing a Mode

- The worst-case input for this algorithm is a list with no equal elements.
- For such a list, its i^{th} element is compared with i − 1 elements of the auxiliary list of distinct values seen so far before being added to the list with a frequency of 1.
- As a result, the worst-case number of comparisons made by this algorithm in creating the frequency list is $\Theta(n^2)$
- The additional n − 1 comparisons needed to find the largest frequency in the auxiliary list do not change the quadratic worst-case efficiency class of the algorithm.
- Efficiency: $\Theta(n^2)$

Example 2: Computing a Mode

Presorting

- As an alternative, let us first sort the input.
- Then all equal values will be adjacent to each other.
- To compute the mode, all we need to do is to find the longest run of adjacent equal values in the sorted array.
- The running time of the algorithm will be dominated by the time spent on sorting since the remainder of the algorithm takes linear time.
- Consequently, with a nlogn sort, this method's worst-case efficiency will be in a better asymptotic class than the worst- case efficiency of the brute-force algorithm.
- Efficiency: $\Theta(n \log n)$

Example 2: Computing a Mode

ALGORITHM *PresortMode*$(A[0..n-1])$

```
//Computes the mode of an array by sorting it first
//Input: An array A[0..n − 1] of orderable elements
//Output: The array's mode
sort the array A
i ← 0                              //current run begins at position i
modefrequency ← 0                  //highest frequency seen so far
while i ≤ n − 1 do
    runlength ← 1;   runvalue ← A[i]
    while i + runlength ≤ n − 1 and A[i + runlength] = runvalue
        runlength ← runlength + 1
    if runlength > modefrequency
        modefrequency ← runlength;   modevalue ← runvalue
    i ← i + runlength
return modevalue
```

Example 3: Searching Problem

- Problem: Search for a given K in A[0..n-1]

Bruteforce

- The brute-force solution here is sequential search, which needs ***n*** comparisons in the worst case.
- Efficiency: $\Theta(n)$

Example 3: Searching Problem

- **Presorting**
- Presorting-based algorithm:
  - Stage 1: Sort the array by an efficient sorting algorithm
  - Stage 2: Apply binary search
- Assuming the most efficient n log n sort, the total running time of such a searching algorithm in the worst case will be

$$T(n) = T_{sort}(n) + T_{search}(n)$$

- Efficiency: $\Theta(n \log n) + \Theta(\log n) = \Theta(n \log n)$

Example 3: Searching Problem

- Brute force algorithm
  Efficiency: $\Theta(n)$
- Presorting-based algorithm
  - Efficiency:$\Theta(n\log n) + \Theta(\log n) = \Theta(n\log n)$
- Good or bad?
- Why do we have our dictionaries, telephone directories, etc. sorted? If we are to search in the same list more than once, the time spent on sorting might well be justified.

Instance simplification

Gaussian Elimination

Gaussian Elimination

- We are certainly familiar with systems of two linear equations in two unknowns:

$$a_{11}x + a_{12}y = b_1$$
$$a_{21}x + a_{22}y = b_2.$$

- Unless the coefficients of one equation are proportional to the coefficients of the other, the system has a unique solution.
- The standard method for finding this solution is to use either equation to express one of the variables as a function of the other and then substitute the result into the other equation, yielding a linear equation whose solution is then used to find the value of the second variable.

Gaussian Elimination (cont.)

- In many applications, we need to solve a system of *n* equations in *n* unknowns:

$$
\begin{aligned}
a_{11}x_1 + a_{12}x_2 + \cdots + a_{1n}x_n &= b_1 \\
a_{21}x_1 + a_{22}x_2 + \cdots + a_{2n}x_n &= b_2 \\
&\vdots \\
a_{n1}x_1 + a_{n2}x_2 + \cdots + a_{nn}x_n &= b_n
\end{aligned}
$$

where n is a large number.

- Theoretically, we can solve such a system by generalizing the substitution method for solving systems of two linear equations.
- However, the resulting algorithm would be extremely cumbersome.

Gaussian Elimination (cont.)

- There is a much more elegant algorithm for solving systems of linear equations called ***Gaussian elimination.***

- The idea of Gaussian elimination is to transform a system of n linear equations in n unknowns to an equivalent system (i.e., a system with the same solution as the original one) with an upper- triangular coefficient matrix, a matrix with all zeros below its main diagonal:

$$\begin{aligned} a_{11}x_1 + a_{12}x_2 + \cdots + a_{1n}x_n &= b_1 \\ a_{21}x_1 + a_{22}x_2 + \cdots + a_{2n}x_n &= b_2 \\ &\vdots \\ a_{n1}x_1 + a_{n2}x_2 + \cdots + a_{nn}x_n &= b_n \end{aligned} \quad \Longrightarrow \quad \begin{aligned} a'_{11}x_1 + a'_{12}x_2 + \cdots + a'_{1n}x_n &= b'_1 \\ a'_{22}x_2 + \cdots + a'_{2n}x_n &= b'_2 \\ &\vdots \\ a'_{nn}x_n &= b'_n. \end{aligned}$$

- In matrix notations, we can write this as $Ax = b \quad \Longrightarrow \quad A'x = b',$

Gaussian Elimination (cont.)

$$A = \begin{bmatrix} a_{11} & a_{12} & \dots & a_{1n} \\ a_{21} & a_{22} & \dots & a_{2n} \\ \vdots & & & \\ a_{n1} & a_{n2} & \dots & a_{nn} \end{bmatrix}, \; b = \begin{bmatrix} b_1 \\ b_2 \\ \vdots \\ b_n \end{bmatrix}, \; A' = \begin{bmatrix} a'_{11} & a'_{12} & \dots & a'_{1n} \\ 0 & a'_{22} & \dots & a'_{2n} \\ \vdots & & & \\ 0 & 0 & \dots & a'_{nn} \end{bmatrix}, \; b = \begin{bmatrix} b'_1 \\ b'_2 \\ \vdots \\ b'_n \end{bmatrix}.$$

- We can easily solve the system with an upper-triangular coefficient matrix by back substitutions as follows.

Gaussian Elimination (cont.)

- First, we can immediately find the value of x_n from the last equation;
- Then we can substitute this value into the next to last equation to get x_{n-1}, and so on,
- Until we substitute the known values of the last n − 1 variables into the first equation, from which we find the value of x_1.

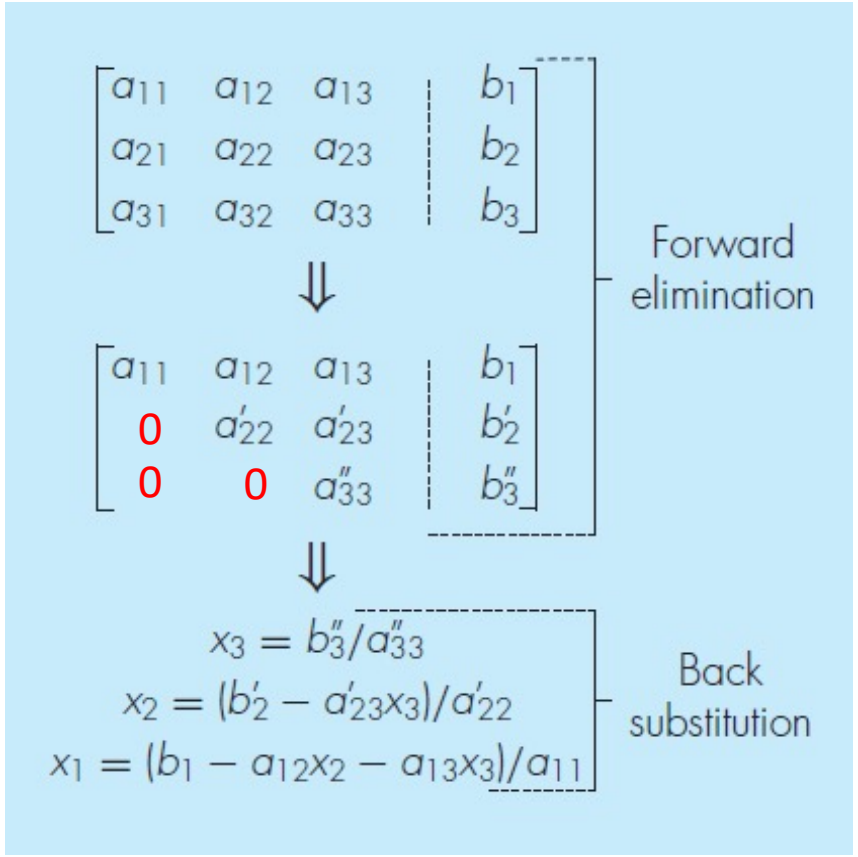
$$
\begin{aligned}
a'_{11}x_1 + a'_{12}x_2 + \cdots + a'_{1n}x_n &= b'_1 \\
a'_{22}x_2 + \cdots + a'_{2n}x_n &= b'_2 \\
&\vdots \\
a'_{nn}x_n &= b'_n.
\end{aligned}
$$

Gaussian Elimination (cont.)

- How can we get from a system with an arbitrary coefficient matrix A to an equivalent system with an upper-triangular coefficient matrix A' ?
- We can do that through a series of the so-called ***elementary operations***:
  - exchanging two equations of the system replacing an equation with its nonzero multiple
  - replacing an equation with a sum or difference of this equation and some multiple of another equation
  - Since no elementary operation can change a solution to a system, any system that is obtained through a series of such operations will have the same solution as the original one.

Gaussian Elimination (cont.)

- Solve the system by Gaussian elimination.



Gaussian Elimination (cont.)

- Let us see how we can get to a system with an upper-triangular matrix.
- First, we use a_{11} as a ***pivot*** to make all x_1 coefficients zeros in the equations below the first one.
- Specifically, we replace the second equation with the difference between it and the first equation multiplied by a_{21}/a_{11} to get an equation with a zero coefficient for x_1.

Gaussian Elimination (cont.)

- Doing the same for the third, fourth, and finally *n*th equation—with the multiples a_{31}/a_{11}, a_{41}/a_{11}, . . . , a_{n1}/a_{11} of the first equation, respectively—makes all the coefficients of x_1 below the first equation zero.
- Then we get rid of all the coefficients of x_2 by subtracting an appropriate multiple of the second equation from each of the equations below the second one.
- Repeating this elimination for each of the first n − 1 variables ultimately yields a system with an upper-triangular coefficient matrix.
- We can operate with just a system's coefficient matrix augmented, as its $(n + 1)^{st}$ column, with the equations' right-hand side values.

Example 1: Gaussian Elimination

- Solve the system by Gaussian elimination.

$$
\begin{aligned}
2x_1 - x_2 + x_3 &= 1 \\
4x_1 + x_2 - x_3 &= 5 \\
x_1 + x_2 + x_3 &= 0.
\end{aligned}
$$

$$
\left[\begin{array}{ccc|c} 2 & -1 & 1 & 1 \\ 4 & 1 & -1 & 5 \\ 1 & 1 & 1 & 0 \end{array}\right] \begin{array}{l} \\ \text{row } 2 - \frac{4}{2} \text{ row } 1 \\ \text{row } 3 - \frac{1}{2} \text{ row } 1 \end{array}
$$

$$
\left[\begin{array}{ccc|c} 2 & -1 & 1 & 1 \\ 0 & 3 & -3 & 3 \\ 0 & \frac{3}{2} & \frac{1}{2} & -\frac{1}{2} \end{array}\right] \begin{array}{l} \\ \\ \text{row } 3 - \frac{1}{2} \text{ row } 2 \end{array}
$$

$$
\left[\begin{array}{ccc|c} 2 & -1 & 1 & 1 \\ 0 & 3 & -3 & 3 \\ 0 & 0 & 2 & -2 \end{array}\right]
$$

Example 1: Gaussian Elimination

Augmented Matrix

$$\left[\begin{array}{ccc|c} 2 & -1 & 1 & 1 \\ 4 & 1 & -1 & 5 \\ 1 & 1 & 1 & 0 \end{array}\right]$$

Upper-triangular coefficient matrix.

$$\left[\begin{array}{ccc|c} 2 & -1 & 1 & 1 \\ 0 & 3 & -3 & 3 \\ 0 & 0 & 2 & -2 \end{array}\right]$$

Example 1: Gaussian Elimination

- Now we can obtain the solution by back substitutions:

$$x_3 = (-2)/2 = -1,$$

$$x_2 = (3 - (-3)x_3)/3 = 0,$$

$$x_1 = (1 - x_3 - (-1)x_2)/2 = 1.$$

Pseudocode of Gaussian Elimination

- Stage 1: Reduction to an upper-triangular matrix

```
for i ← 1 to n-1 do
    for j ← i+1 to n do
        for k ← i to n+1 do
            A[j, k] ← A[j, k] - A[i, k] * A[j, i] / A[i, i]
```

- Stage 2: Back substitutions

```
for j ← n down to 1 do
    t ← 0
    for k ← j +1 to n do
        t ← t + A[j, k] * x[k]
    x[j] ← (A[j, n+1] - t) / A[j, j]
```

- Efficiency: $\Theta(n^3) + \Theta(n^2) = \Theta(n^3)$

Georgia State University

CSC 4520/6520
Design & Analysis of Algorithms

CHAPTER 6: TRANSFORM-AND-CONQUER (AVL)

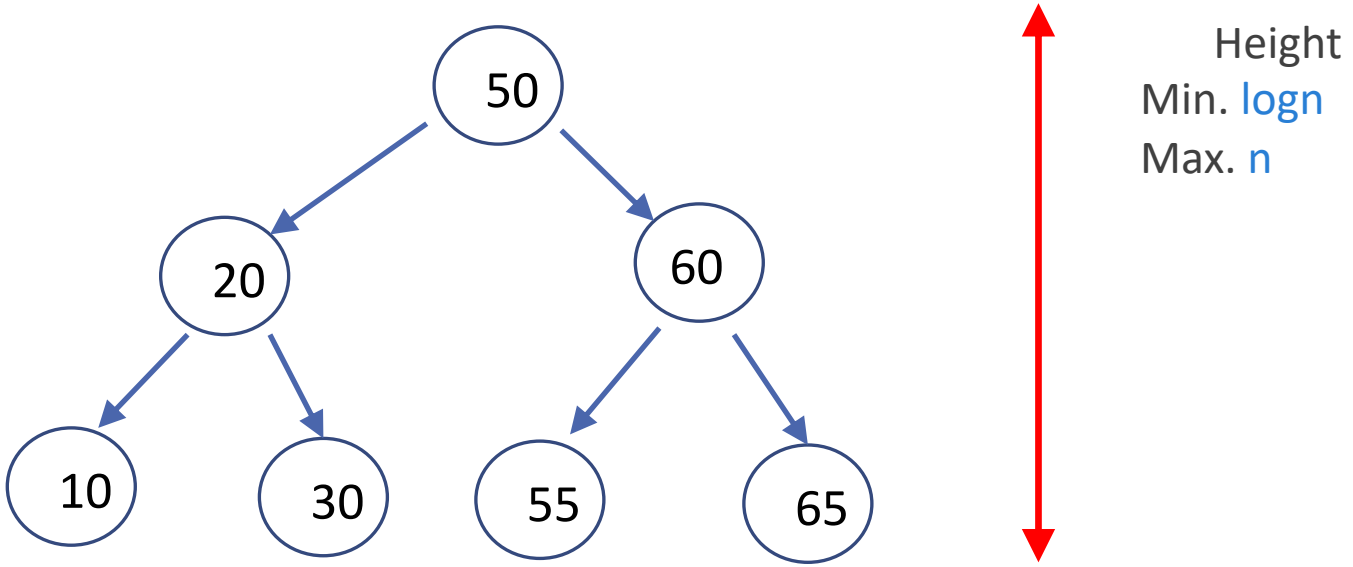
Abdullah Bal, PhD
Spring 2024

Taxonomy of Searching Algorithms

- List searching
  - sequential search
  - binary search
  - interpolation search
- Tree searching
  - binary search tree
  - binary balanced trees: AVL trees, red-black trees
  - multiway balanced trees: 2-3 trees, 2-3-4 trees, B trees
- Hashing
  - open hashing (separate chaining)
  - closed hashing (open addressing)

Binary Search Trees

- Binary search tree is a binary tree whose nodes contain elements of a set of orderable items,
  - One element per node,
  - All elements in the left subtree are smaller than the element in the subtree's root,
  - All the elements in the right subtree are greater than it.

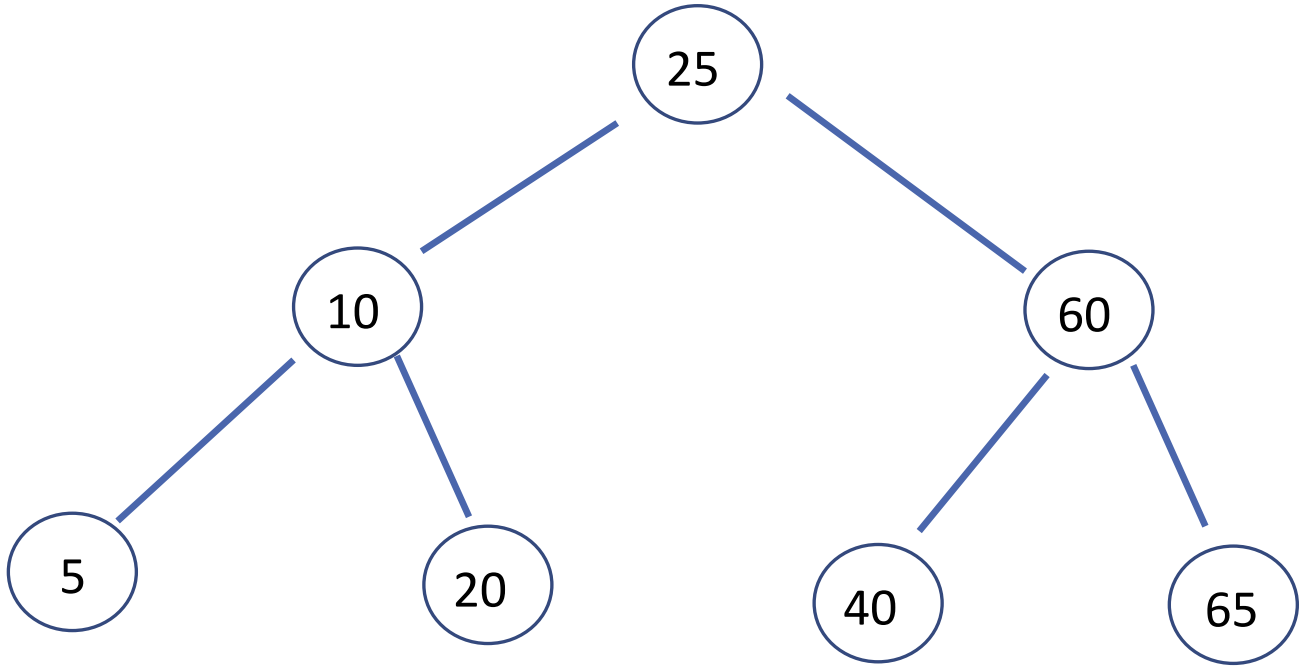


Example: Binary Search Tree

Keys: 25, 60, 10, 65, 20, 5, 40

Example: Binary Search Tree

Keys: 25, 60, 10, 65, 20, 5, 40



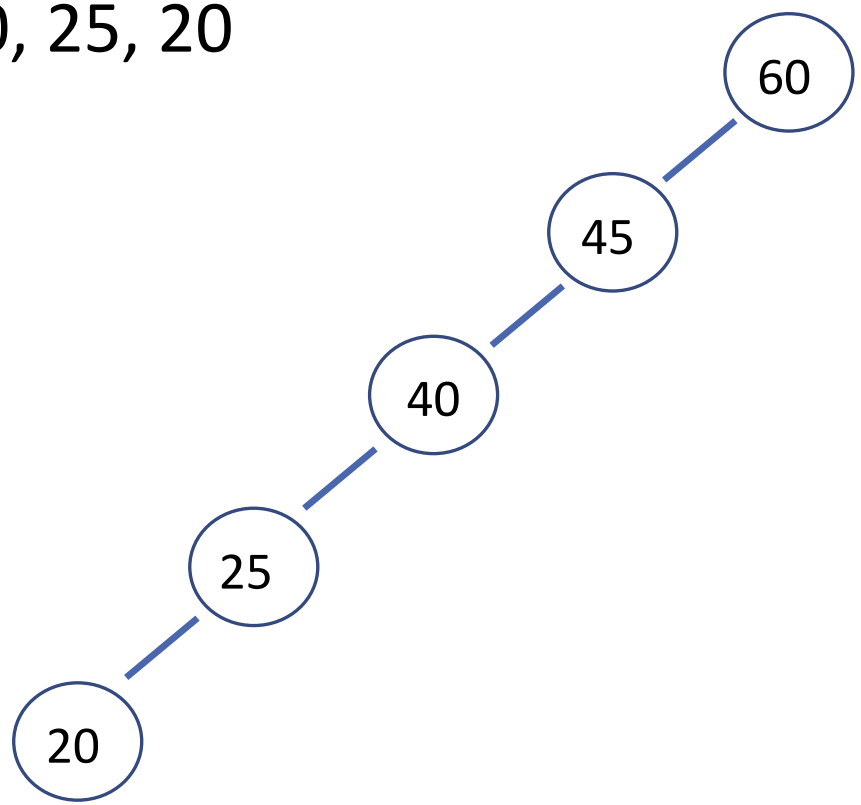
Height:
logn

Example: Binary Search Tree

Keys: 60, 45, 40, 25, 20, 10, 5

Example: Binary Search Tree

Keys: 60, 45, 40, 25, 20



Height

n

Binary Search Trees

- This transformation from a set to a binary search tree is an example of the representation-change technique.
- We gain in the time efficiency of searching, insertion, and deletion, which are all in $\Theta(\log n)$, but only in the average case.
- In the worst case, these operations are in $\Theta(n)$ because the tree can degenerate into a severely unbalanced one with its height equal to $n - 1$.

Binary Search Trees

- Computer scientists have expended a lot of effort in trying to find a structure that preserves the good properties of the classical binary search tree
- Principally, the logarithmic efficiency of the dictionary operations avoiding its worst-case degeneracy.
- They have come up with two approaches:
  - Instance-simplification variety
  - Representation-change variety

SELF-BALANCING AVL TREE

The Instance-simplification Variety (AVL)

- An unbalanced binary search tree is transformed into a balanced one.
- Such trees are called ***self-balancing***.
- An ***AVL tree*** requires the difference between the heights of the left and right subtrees of every node never exceed 1.
- If an insertion or deletion of a new node creates a tree with a violated balance requirement, the tree is restructured by one of a family of special transformations called ***rotations*** that restore the balance required.

The Representation-change Variety (2-3 Trees)

- Allow more than one element in a node of a search tree.
- Specific cases of such trees are ***2-3 trees***, ***2-3-4 trees***, and more general and important ***B-trees***.
- They differ in the number of elements admissible in a single node of a search tree, but all are perfectly balanced.
- We discuss the simplest case of such trees, the 2-3 tree, in this section, leaving the discussion of *B*-trees for Chapter 7.

Balanced Search Trees

- Attractiveness of binary search tree is marred by the worst-case efficiency. Two ideas to overcome it are:
- To rebalance binary search tree when a new insertion makes the tree "too unbalanced"
  - AVL trees
  - Red-black trees
- To allow more than one key per node of a search tree
  - 2-3 trees
  - 2-3-4 trees
  - B-trees

Balanced trees: AVL

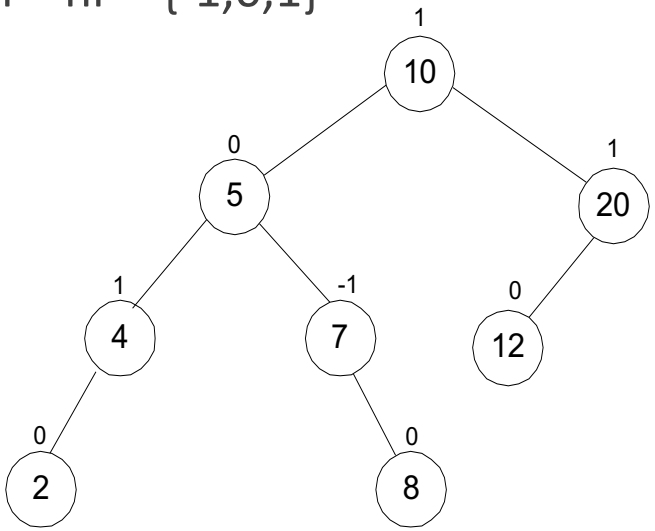
- AVL (**A**delson-**V**elsky-**L**andis) trees were invented in 1962 by two Russian scientists, G. M. Adelson-Velsky and E. M. Landis, after whom this data structure is named.

AVL trees

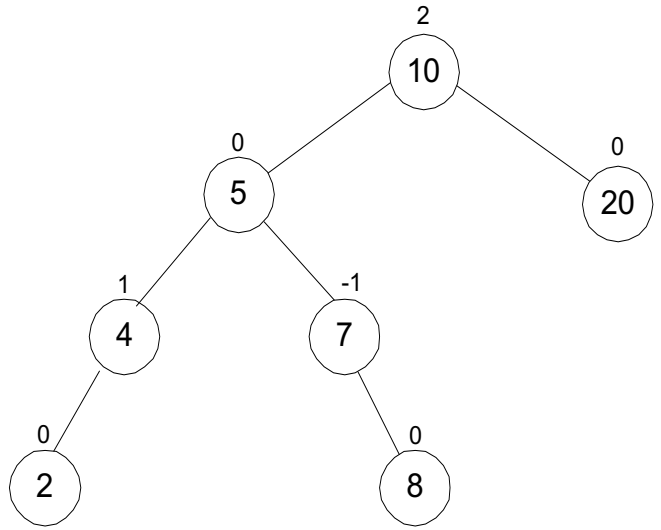
- **An *AVL tree* is a binary search tree** in which the ***balance factor*** of every node, which is defined as the difference between the heights of the node's left and right subtrees, is either 0 or +1 or −1.

balance factor = height of the left subtree – height of the right subtree

balance factor = hl – hr = {-1,0,1}



AVL Tree



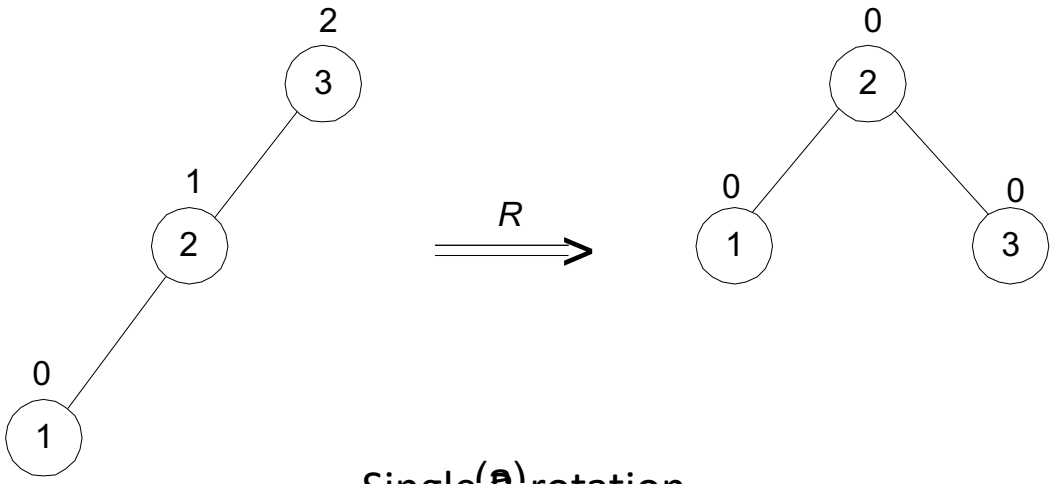
Not AVL Tree

Rotations

- If an insertion of a new node makes an AVL tree unbalanced, we transform the tree by a rotation.
- If there are several such nodes, we rotate the tree rooted at the unbalanced node that is the closest to the newly inserted leaf.
- There are only four types of rotations; in fact, two of them are mirror images of the other two.

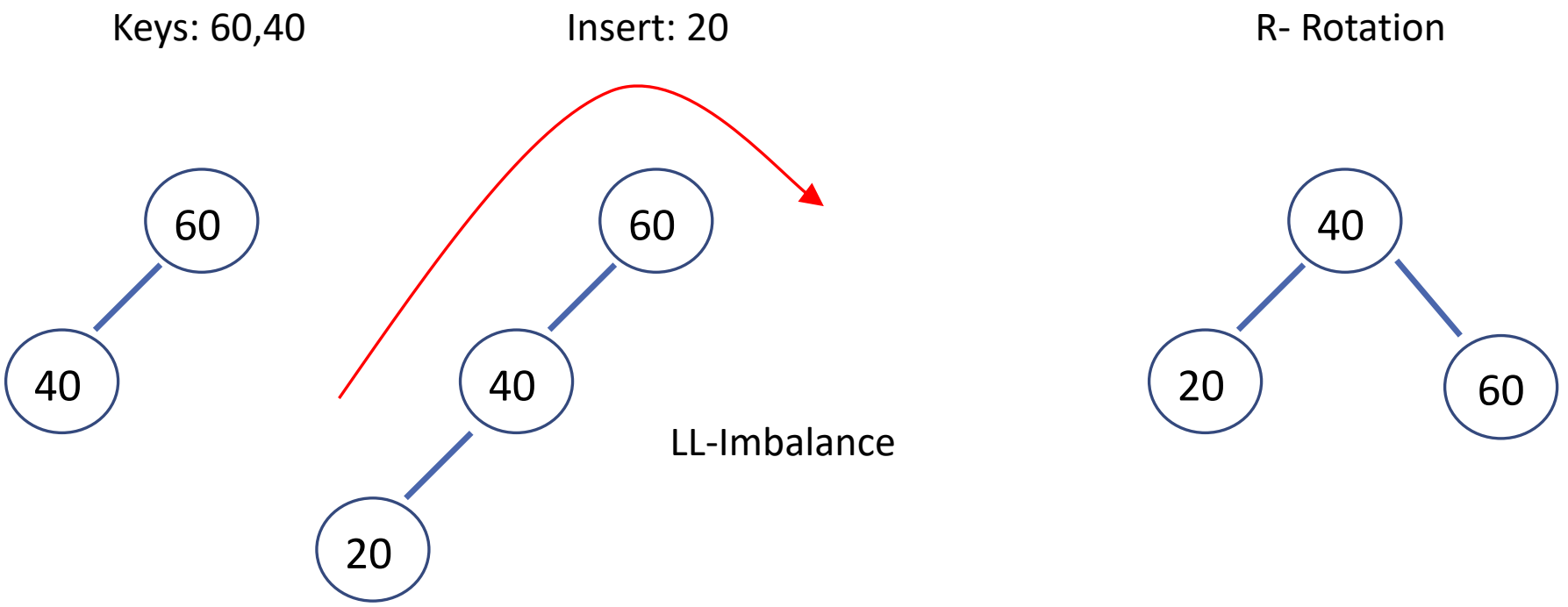
Single Right Rotation (R-Rotation)

- The first rotation type is called the ***single right rotation***, or ***R-rotation***.
- Note that this rotation is performed after a new key is inserted into the left subtree of the left child of a tree whose root had the balance of +1 before the insertion.



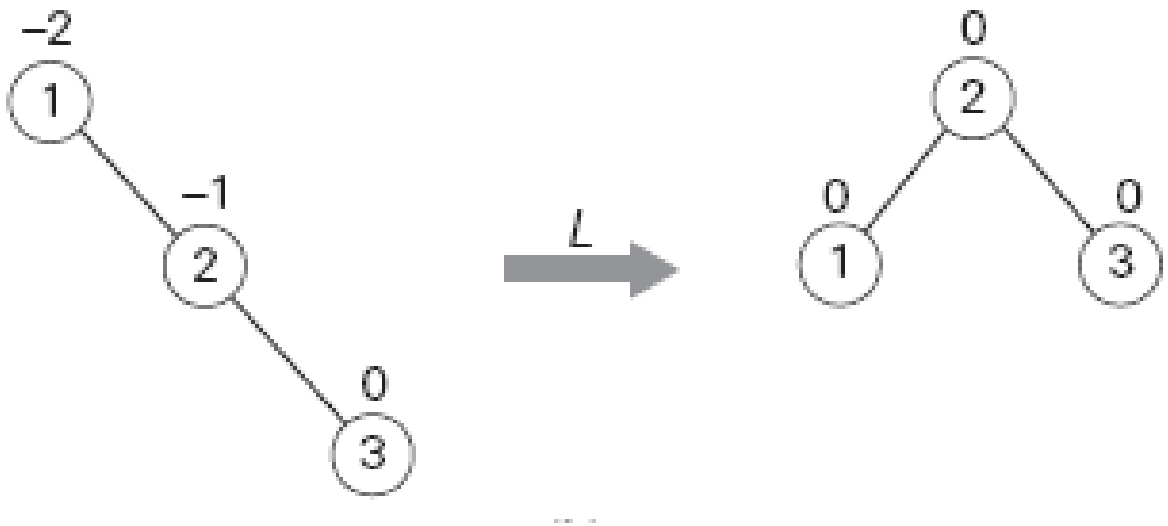
Single R-rotation (a)

Example: Single R Rotation



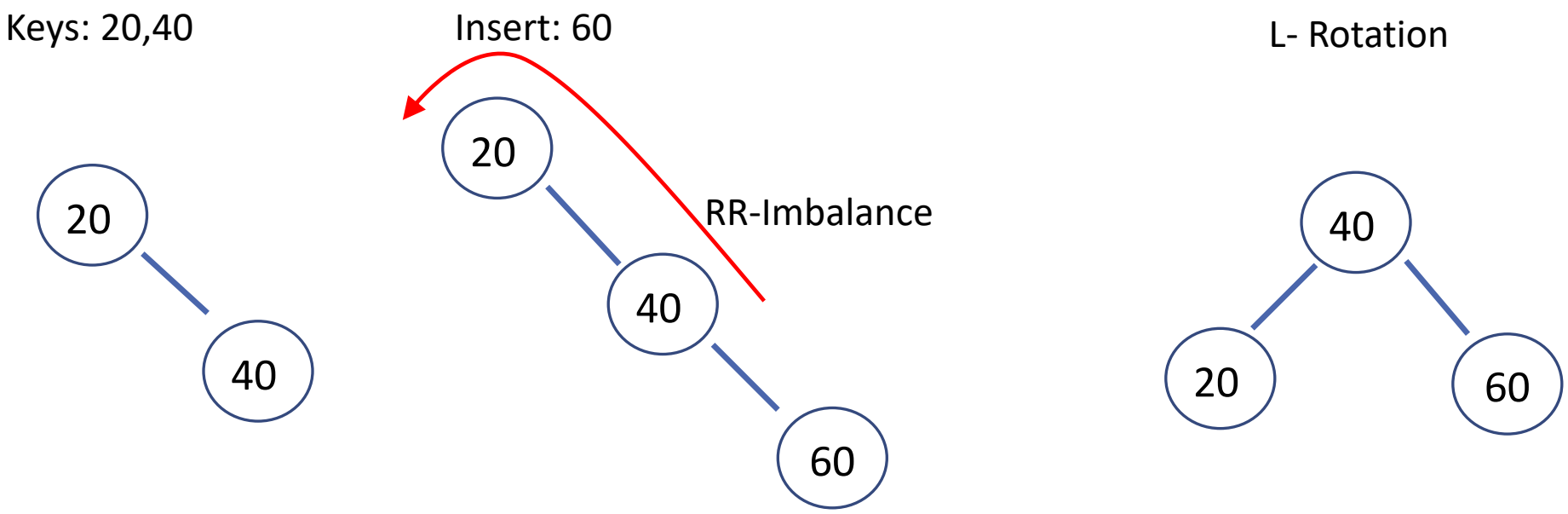
Single Left Rotation

- The symmetric ***single left rotation***, or ***L-rotation***, is the mirror image of the single R-rotation.
- It is performed after a new key is inserted into the right subtree of the right child of a tree whose root had the balance of −1 before the insertion.



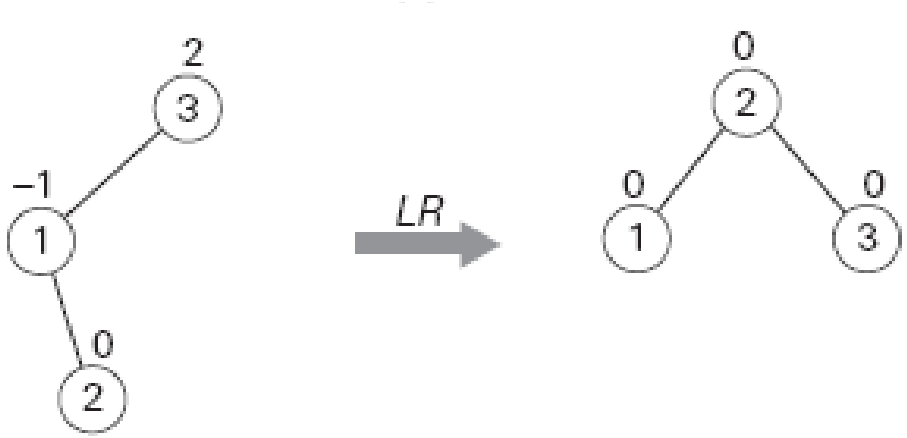
Single L-rotation

Example: Single L Rotation



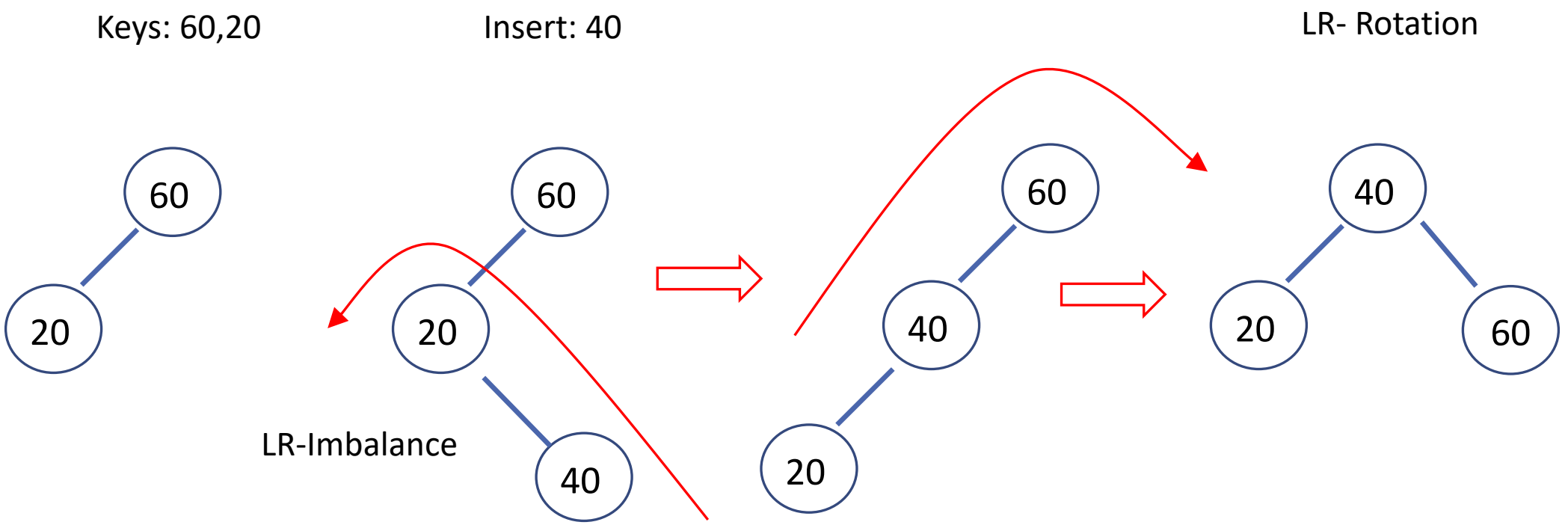
Double Left-Right (LR) Rotation

- The second rotation type is called the ***double left-right rotation*** (***LR- rotation***).
- We perform the *L*-rotation of the left subtree of root r followed by the *R*-rotation of the new tree rooted at r.
- It is performed after a new key is inserted into the right subtree of the left child of a tree whose root had the balance of +1 before the insertion.



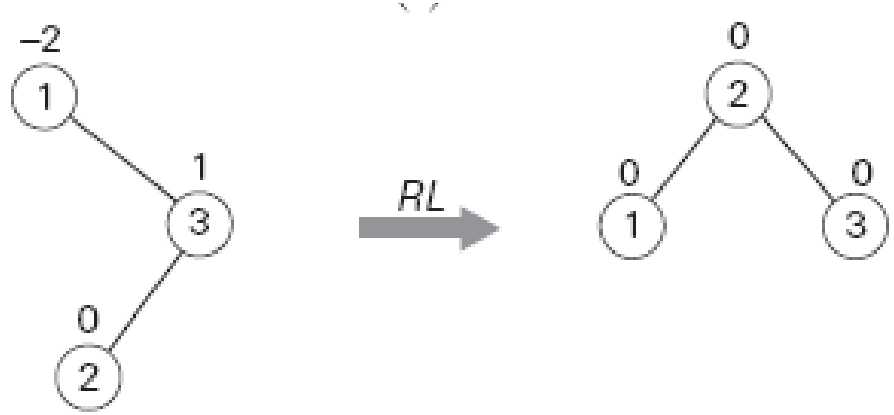
Double LR-rotation

Example: Double LR Rotation



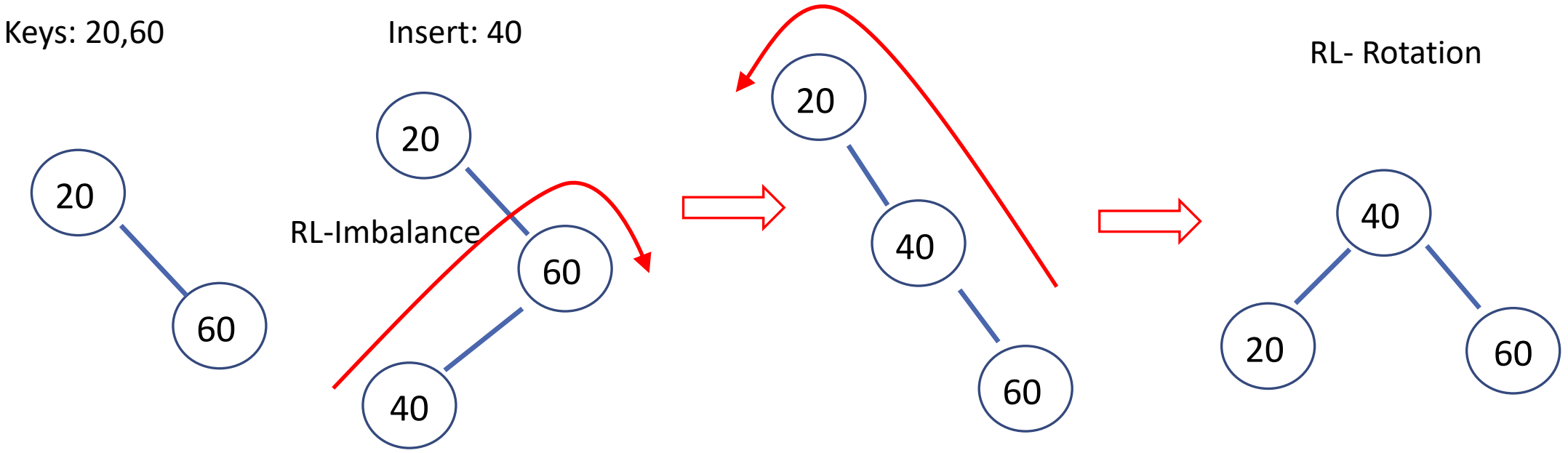
Double Right-Left (RL) Rotation

- The ***double right-left rotation*** (***RL-rotation***) is the mirror image of the double LR-rotation.
- We perform the *L*-rotation of the left subtree of root r followed by the *R*-rotation of the new tree rooted at r.
- It is performed after a new key is inserted into the right subtree of the left child of a tree whose root had the balance of +1 before the insertion.



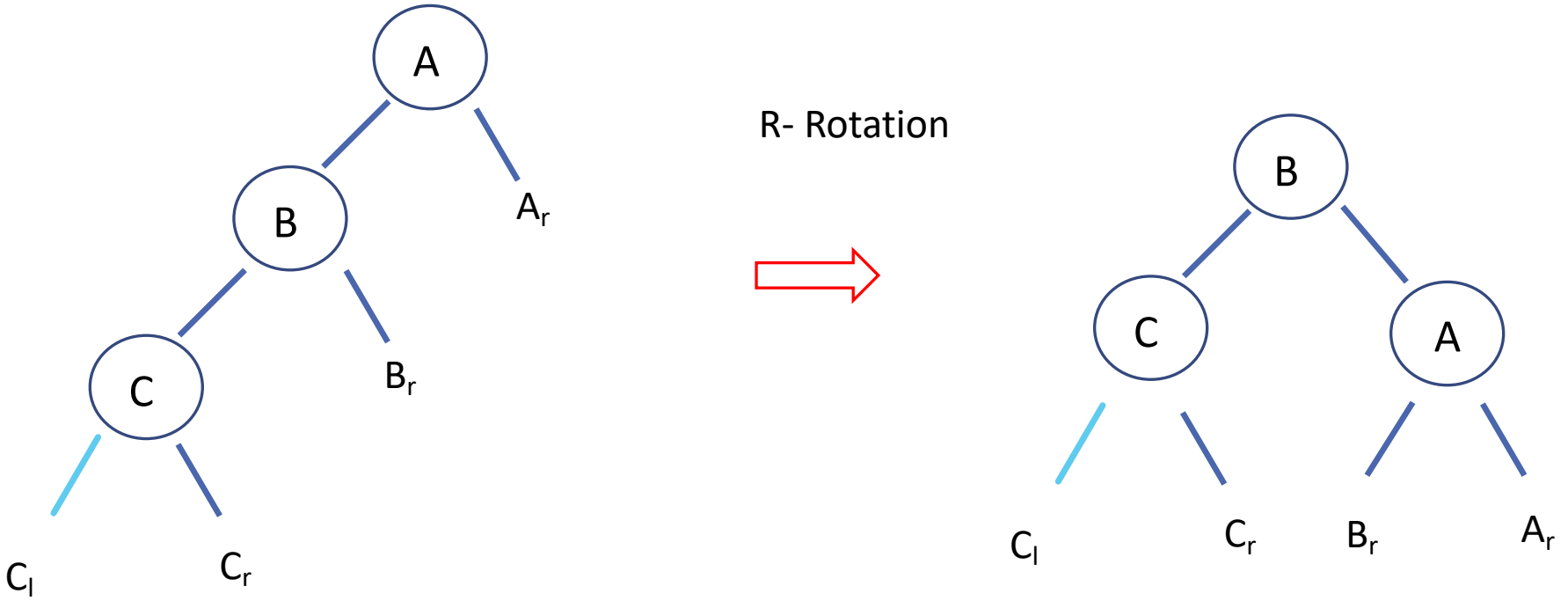
Double RL-rotation

Example: Double RL Rotation



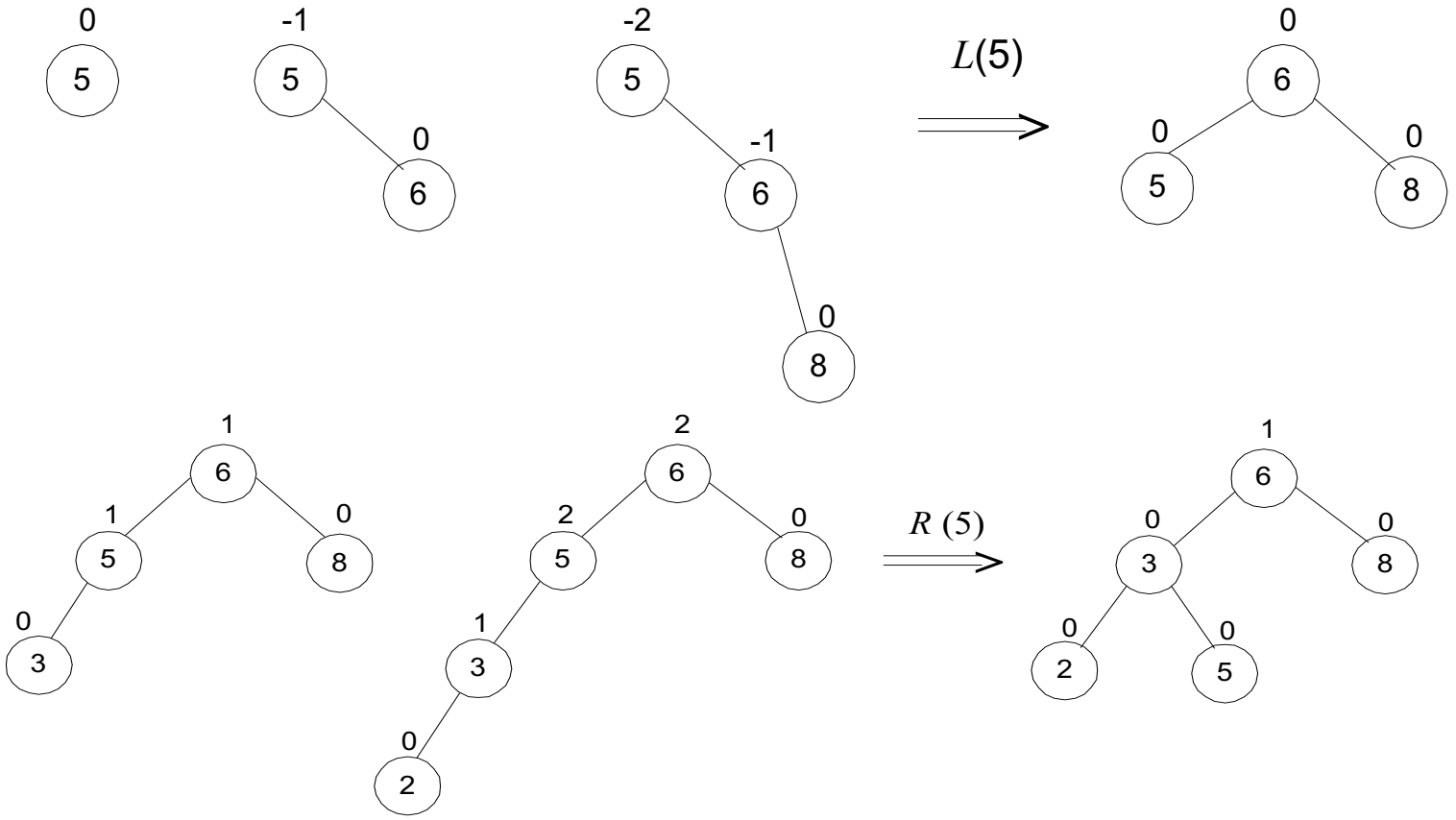
General Case: Single R Rotation

- Rotations guarantee that a resulting tree is balanced, but they should also preserve the basic requirements of a binary search tree.
- And the same relationships among the key values hold, as they must, for the balanced tree after the rotation.



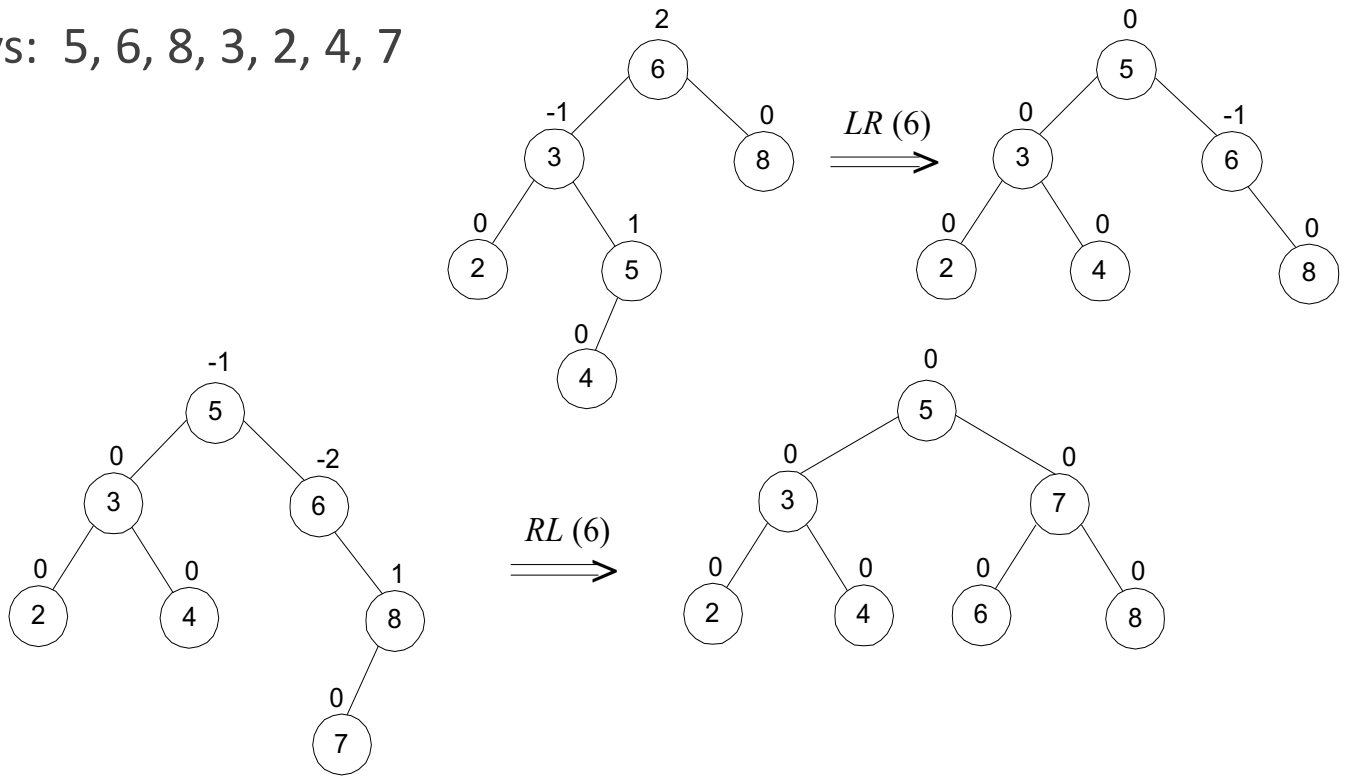
Example 1: AVL tree construction

- Construct an AVL tree for the list 5, 6, 8, 3, 2, 4, 7



Example 1: AVL tree construction (cont.)

- If there are several nodes with the ±2 balance, the rotation is done for the tree rooted at the unbalanced node that is the closest to the newly inserted leaf.
- Keys: 5, 6, 8, 3, 2, 4, 7

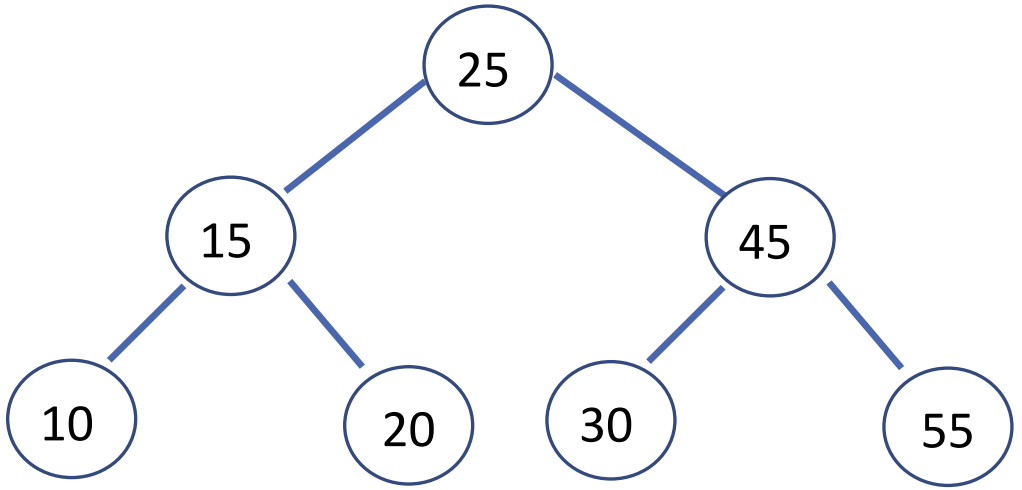


Example 2: AVL tree construction

Construct an AVL tree for the list 45, 15, 10, 25, 30, 20, 55

Example 2: AVL Tree

AVL tree for the list 45, 15, 10, 25, 30, 20, 55



Analysis of AVL trees

- $h \leq 1.4404 \log_2 (n + 2) - 1.3277$
- average height: $1.01 \log_2 n + 0.1$ for large n (found empirically)

- Search and insertion are O(log n)

- Deletion is more complicated but is also O(log n)
- Disadvantages:
  - frequent rotations
  - Complexity
- A similar idea: *red-black trees*

Georgia State University

CSC 4520/6520
Design & Analysis of Algorithms

CHAPTER 6: TRANSFORM-AND-CONQUER (REPRESENTATION CHANGE-2_3 TREE)

Abdullah Bal, PhD

2-3 Tree

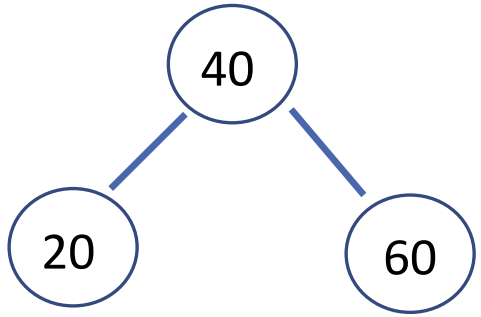
- The simplest implementation of multiway search trees is **2-3 trees**, introduced by the U.S. computer scientist John Hopcroft in 1970.
- A ***2-3 tree*** is a tree that can have nodes of two kinds:
  - 2-nodes
  - 3-nodes.

2-3 Tree Properties

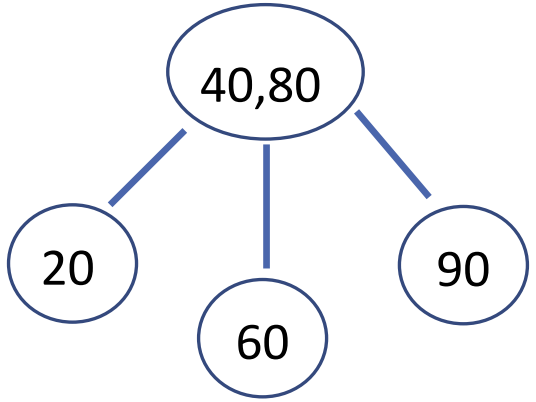
1. Each node can either be a leaf node, 2- node, or 3-node
2. Keys is stored in sorted order.
3. It is a balanced tree.
4. All the leaf nodes are at same level.
5. Always insertion is done at leaf.

1 Leaf node

1 Key, 2 Children

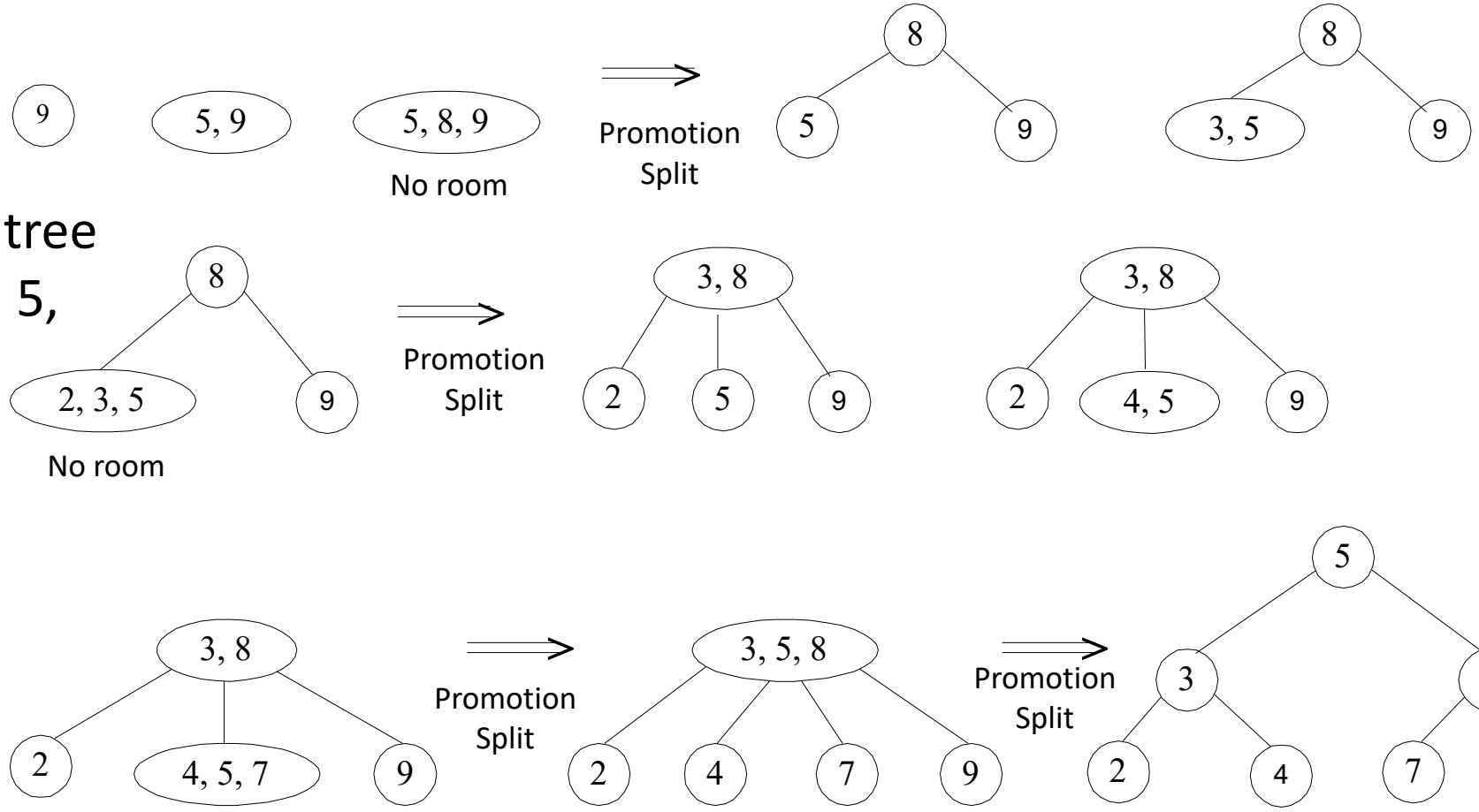


2 Keys, 3 Children



Example 1: 2-3 tree construction

- Construct a 2-3 tree for the list of 9, 5, 8, 3, 2, 4, 7



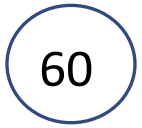
Example 2: 2-3 tree construction

Construct a 2-3 tree for the list of 60, 35, 20, 70, 65, 50, 10

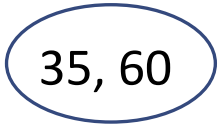
Example 2: 2-3 tree construction

Construct a 2-3 tree for the list of 60, 35, 20, 70, 65, 50, 10

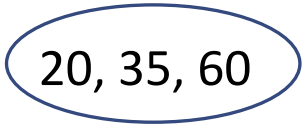
1-



2-

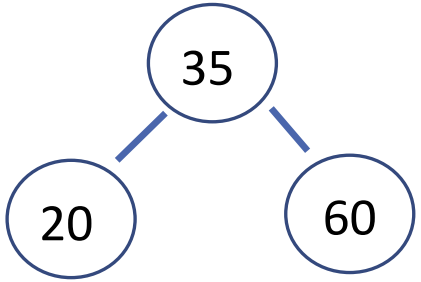


3-

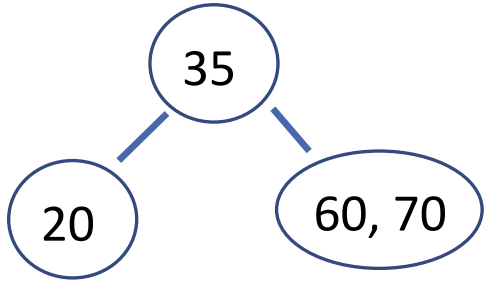


No room

Promotion
Split



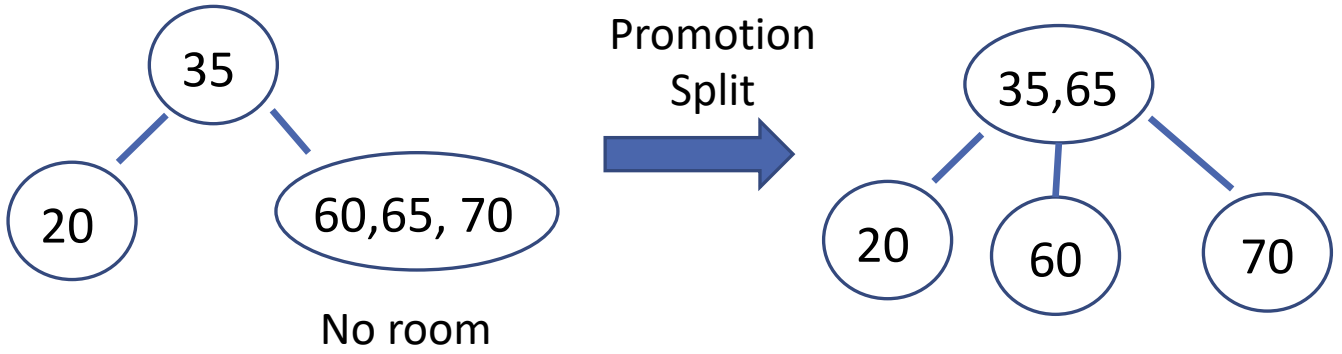
4-



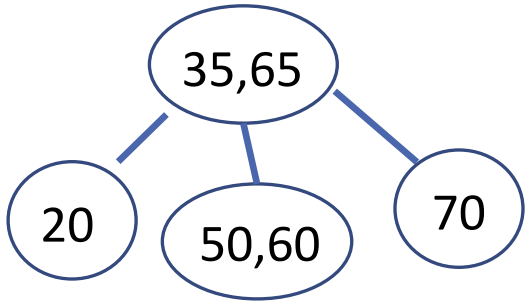
Example 2: 2-3 tree construction

Construct a 2-3 tree for the list of 60, 35, 20, 70, 65, 50, 10

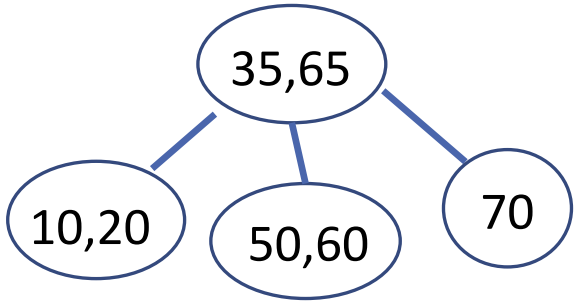
5-



6-



7-

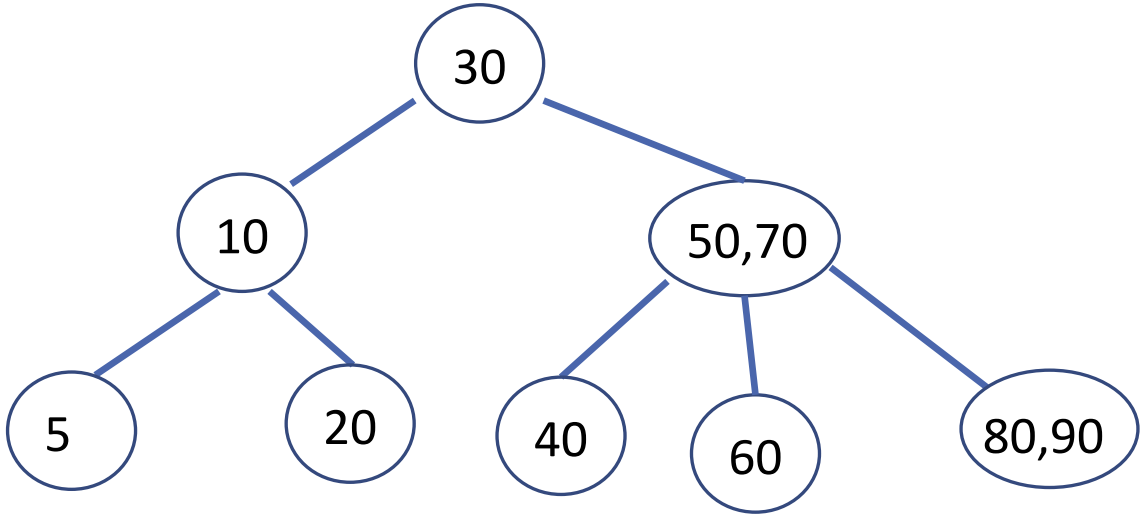


Example 3: 2-3 tree construction

Construct a 2-3 tree for the list of 40, 50, 60, 30, 20, 10,5, 70, 80,90

Example 3: 2-3 tree construction

- Construct a 2-3 tree for the list of 40, 50, 60, 30, 20, 10,5, 70, 80,90
- Solution:



Analysis of 2-3 trees

- As for any search tree, the efficiency of the dictionary operations depends on the tree's height.
- A 2-3 tree of height h with the smallest number of keys is a full tree of 2-nodes (such as the final tree in the previous figure for h = 2).
- Therefore, for any 2-3 tree of height h with n nodes, we get the inequality

$$n \geq 1 + 2 + \cdots + 2^h = 2^{h+1} - 1,$$
$$h \leq \log_2(n+1) - 1.$$

- On the other hand, a 2-3 tree of height h with the largest number of keys is a full tree of 3-nodes, each with two keys and three children.

$$n \leq 2 \cdot 1 + 2 \cdot 3 + \cdots + 2 \cdot 3^h = 2(1 + 3 + \cdots + 3^h) = 3^{h+1} - 1$$

- Therefore, for any 2-3 tree with n nodes, $h \geq \log_3(n+1) - 1.$

Analysis of 2-3 trees

- These lower and upper bounds on height h,

$$\log_3(n+1) - 1 \leq h \leq \log_2(n+1) - 1,$$

- The time efficiencies of searching, insertion, and deletion are all in Θ(log n) in both the worst and average case.