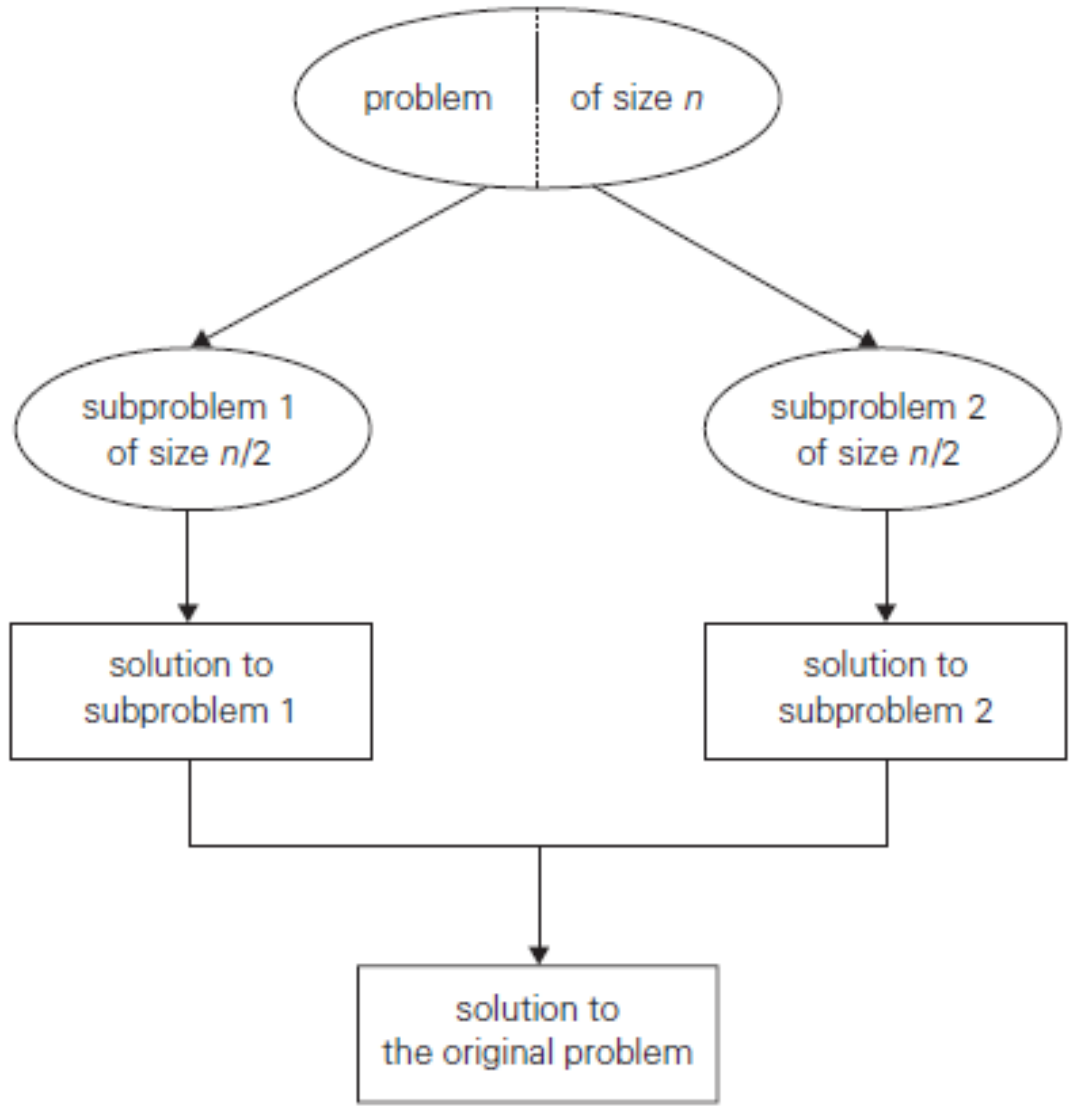
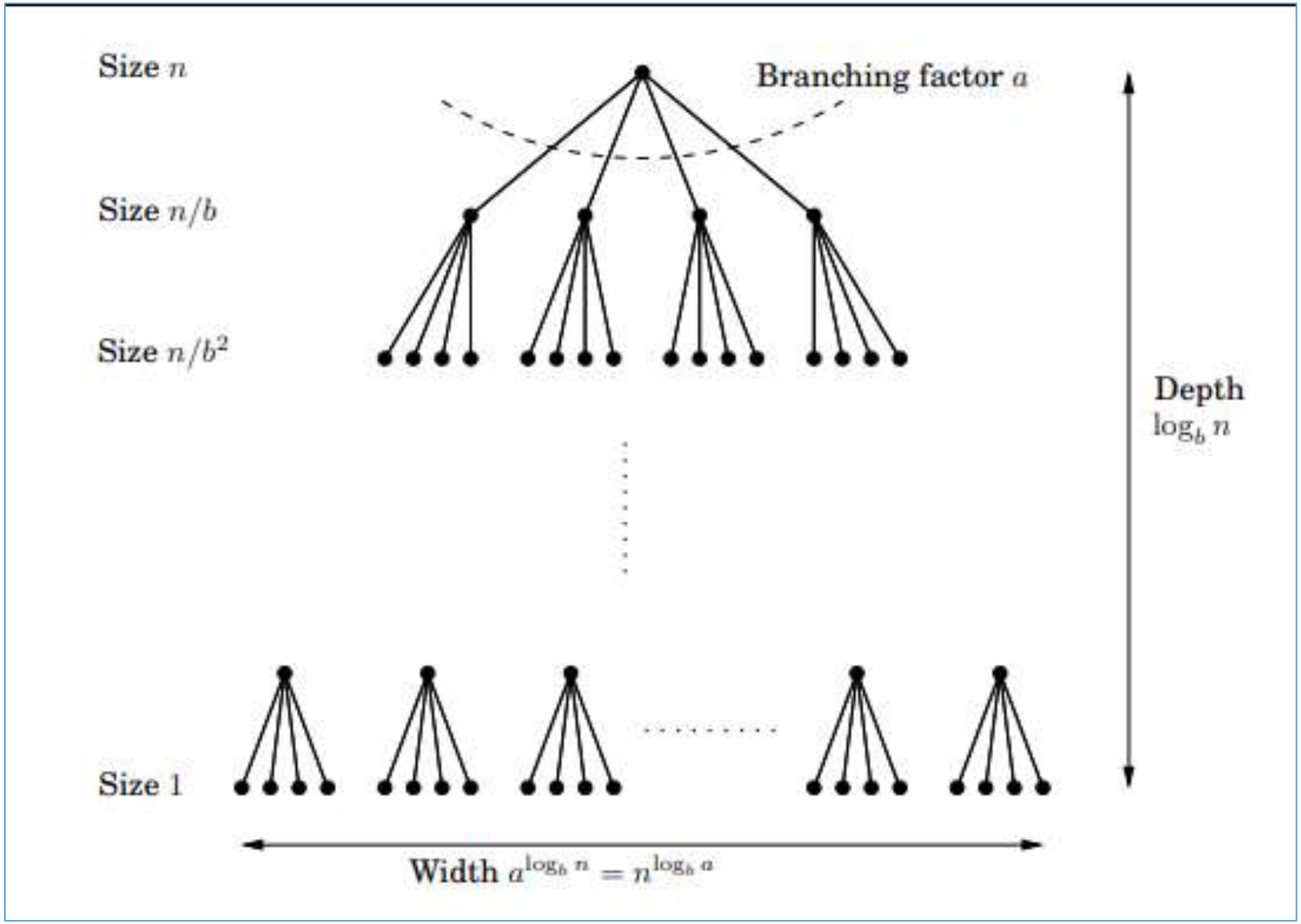
Divide and Conquer

Divide-and-conquer

- **D**ivide-and-conquer is probably the best-known general algorithm design technique.
- Divide-and-conquer algorithms work according to the following general plan:
  - A problem is divided into several sub problems of the same type, ideally of about equal size.
  - Recursively solve the sub problems
  - The solutions to the sub problems are combined to get a solution to the original problem.


problem
of size n
subproblem 1
of size n/2
subproblem 2
of size n/2
solution to
subproblem 1
solution to
subproblem 2
solution to
the original problem

Each problem of size n is divided into a sub-problems of size n/b.



- Divide-and-conquer algorithms often follow a generic pattern
- They tackle a problem of size n by recursively solving
  - “a” subproblems of size “n/b” and
  - then combining these answers in $O(n^d)$ time
  - for some a, b, d > 0

Assuming that size *n* is a power of *b* to simplify our analysis, we get the following recurrence for the running time *T (n)*:

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

$$T(n) = aT(n/b) + f(n),$$

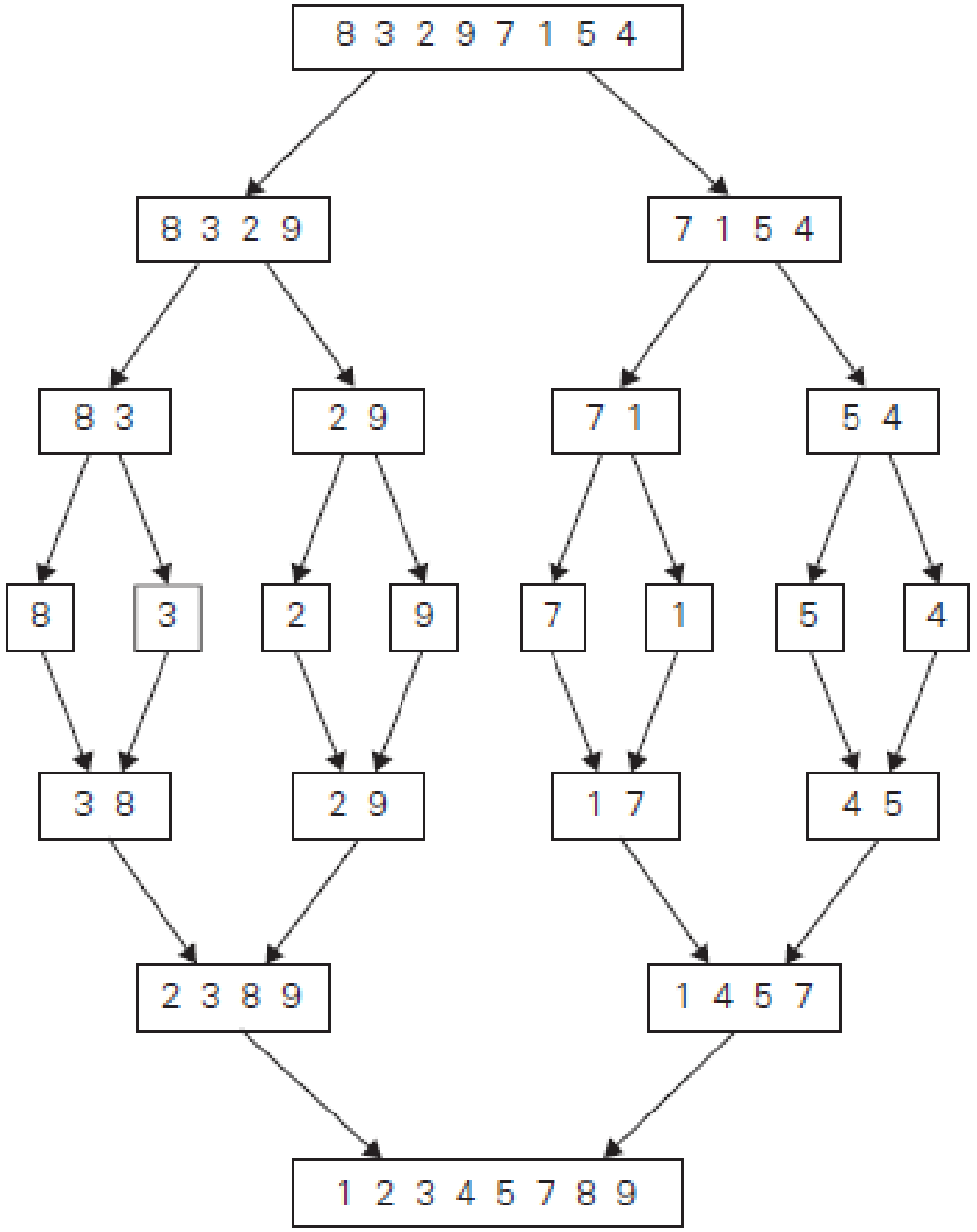
Master Theorem If $f(n) \in \Theta(n^d)$ where $d \geq 0$ in recurrence (5.1), then

$$T(n) \in \begin{cases} \Theta(n^d) & \text{if } a < b^d, \\ \Theta(n^d \log n) & \text{if } a = b^d, \\ \Theta(n^{\log_b a}) & \text{if } a > b^d. \end{cases}$$

Merge Sort

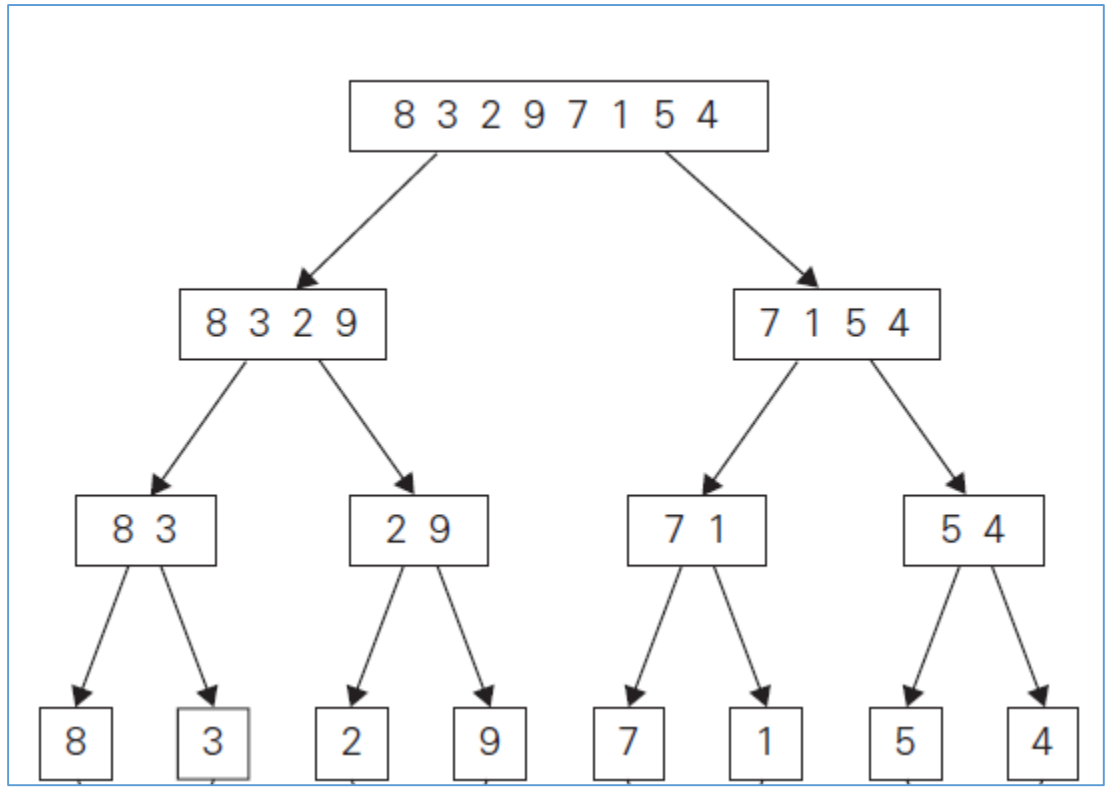
- Mergesort is a perfect example of a successful application of the divide-and conquer technique.
- It sorts a given array $A[0..n-1]$ by dividing it into two halves

  $A[0..n/2-1]$ and $A[n/2..n-1]$, sorting each of them recursively, and then
- Merging the two smaller sorted arrays into a single sorted one.


8 3 2 9 7 1 5 4
8 3 2 9
7 1 5 4
8 3
2 9
7 1
5 4
8
3
2
9
7
1
5
4
3 8
2 9
1 7
4 5
2 3 8 9
1 4 5 7
1 2 3 4 5 7 8 9

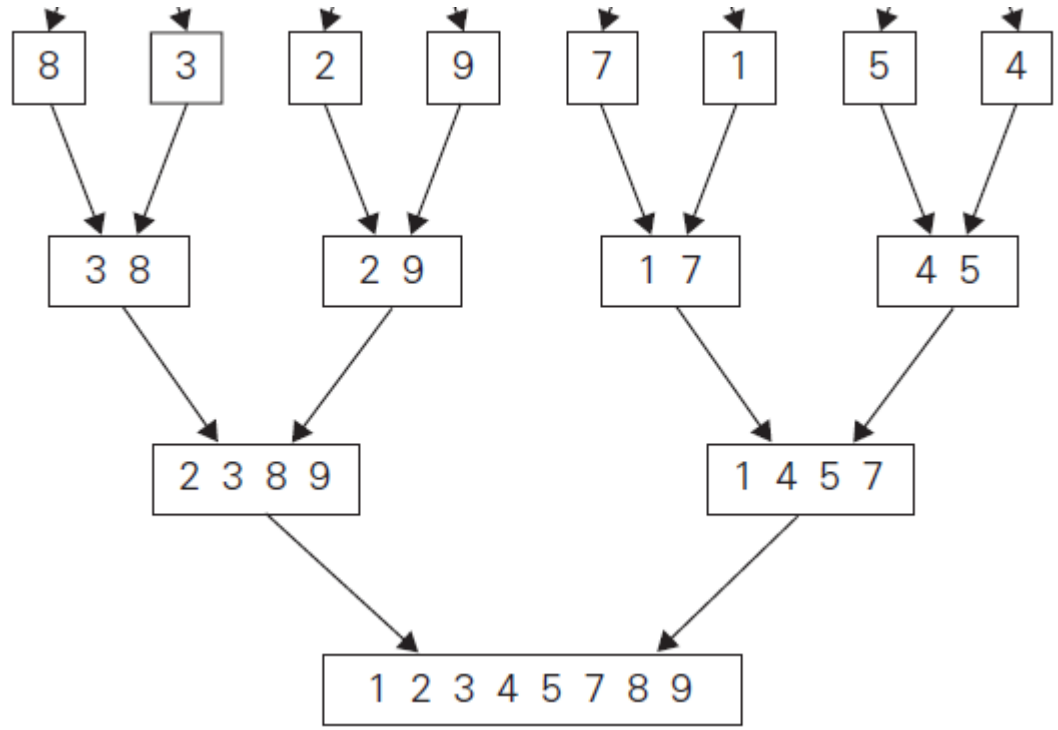
ALGORITHM *Mergesort*$(A[0..n-1])$

//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 $A[0..\lfloor n/2 \rfloor - 1]$ to $B[0..\lfloor n/2 \rfloor - 1]$
    copy $A[\lfloor n/2 \rfloor..n-1]$ to $C[0..\lceil n/2 \rceil - 1]$
    Mergesort$(B[0..\lfloor n/2 \rfloor - 1])$
    Mergesort$(C[0..\lceil n/2 \rceil - 1])$
    Merge(B, C, A)



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 $B[i] \le C[j]$
        $A[k] \leftarrow B[i];\ i \leftarrow i+1$
    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 $A[k..p+q-1]$



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

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

- $C_{merge}(n)$, 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.
- In the worst case, neither of the two arrays becomes empty before the other one contains just one element
- Therefore, for the **worst case, $C_{merge}(n) = n - 1$**, and we have the recurrence

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

Question

- Analyze the time complexity to sort an array using Merge Sort in its worst case.

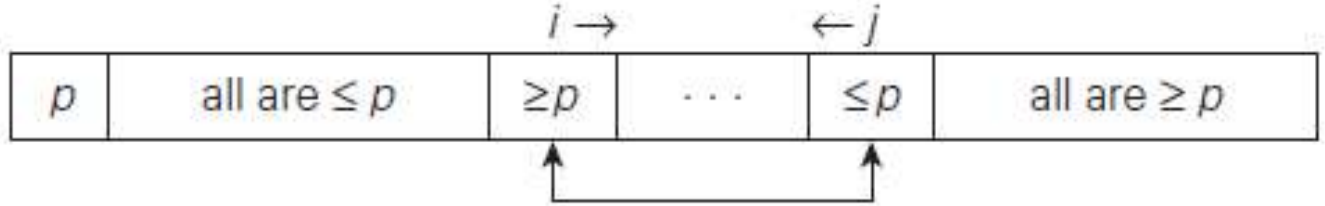
Quick Sort

- QuickSort is a sorting algorithm based on the Divide and Conquer algorithm
- That picks an element as a pivot and partitions the given array around the picked pivot by placing the pivot in its correct position in the sorted array.

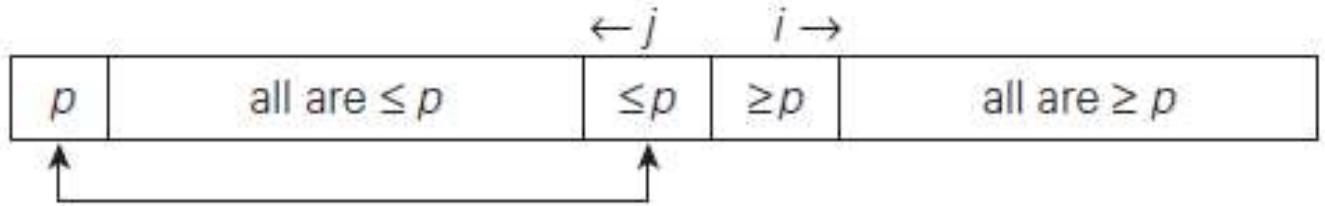
$$\underbrace{A[0] \ldots A[s-1]}_{\text{all are} \leq A[s]} \quad A[s] \quad \underbrace{A[s+1] \ldots A[n-1]}_{\text{all are} \geq A[s]}$$

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

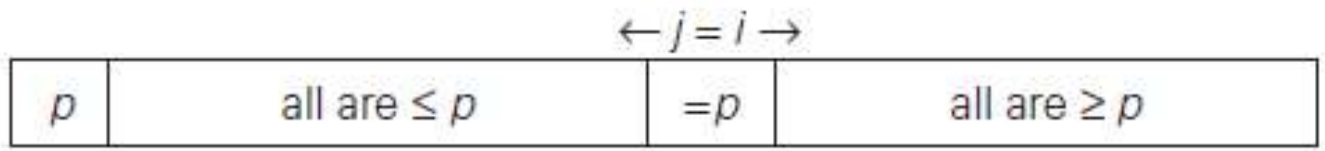
After both scans stop, three situations may arise, depending on whether or not the scanning indices have crossed. If scanning indices i and j have not crossed, i.e., $i < j$, we simply exchange $A[i]$ and $A[j]$ and resume the scans by incrementing i and decrementing j, respectively:



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



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 (why?). Thus, we have the subarray partitioned, with the split position $s = i = j$:



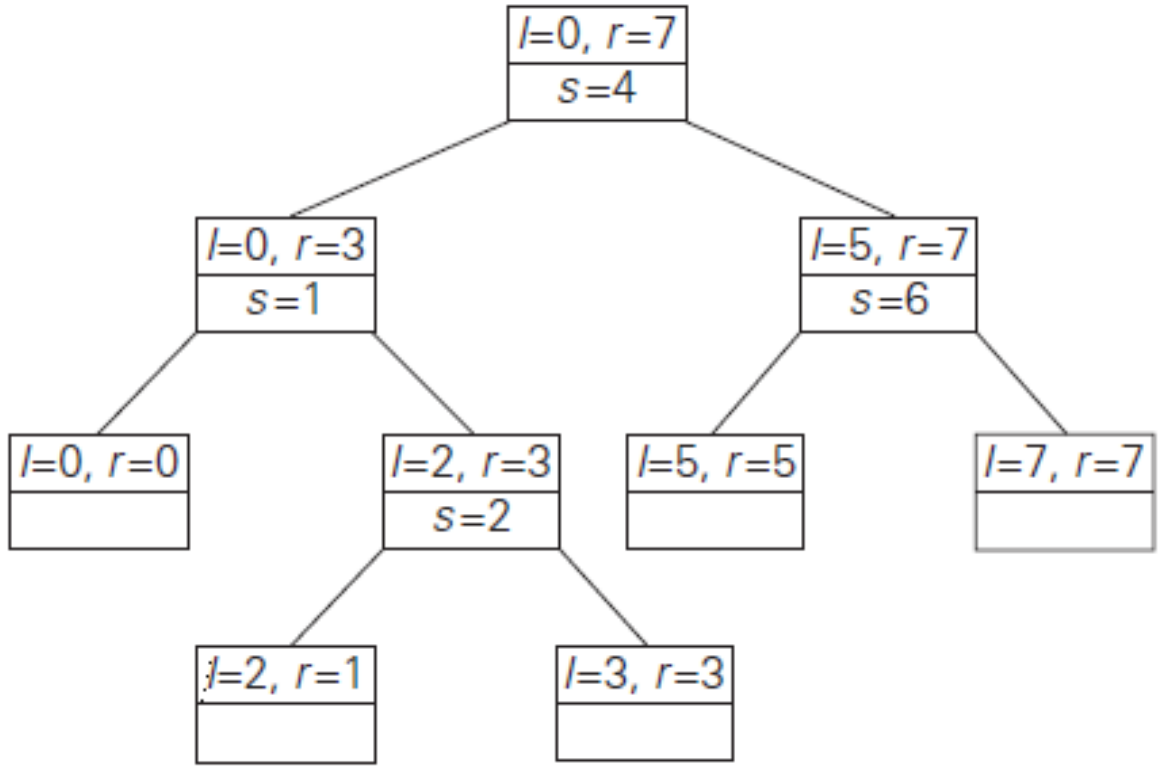
ALGORITHM *Quicksort*$(A[l..r])$

//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$(A[s+1..r])$

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 ← A[l]
i ← l; j ← r + 1
repeat
    repeat i ← i + 1 until A[i] ≥ p
    repeat j ← j − 1 until A[j] ≤ p
    swap(A[i], A[j])
until i ≥ j
swap(A[i], A[j])   //undo last swap when i ≥ j
swap(A[l], A[j])
return j
```

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



The number of key comparisons made before a partition is achieved is $n + 1$ if the scanning indices cross over and n if they coincide

Best Case : If all the splits happen in the middle of corresponding subarrays, we will have the best case. The number of key comparisons in the best case satisfies the recurrence

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

As Per Master's Theorem

$$C_{best}(n) = n \log_2 n.$$

Worst Case

- In the worst case, all the splits will be skewed to the extreme: one of the two subarrays will be empty, and the size of the other will be just 1 less than the size of the subarray being partitioned.
- This unfortunate situation will happen, in particular, for increasing arrays, i.e., for inputs for which the problem is already solved! Indeed, if $A[0..n-1]$ is a strictly increasing array and we use $A[0]$ as the pivot, the left-to-right scan will stop on $A[1]$ while the right-to-left scan will go all the way to reach $A[0]$, indicating the split at position 0

Question

- Write quick sort algorithm to sort an array. Find its time complexity. Apply the algorithm to sort the following list of elements.

{55, 28, 35, 78, 110, 48, 88}

Questions

- Apply Stressen's matrix multiplication to multiply the following two matrices:

1	2	3	4
2	3	4	1
3	4	1	2
4	1	2	3

4	3	2	1
1	4	3	2
2	1	4	3
3	1	2	4

Multiplication of Large Integers.

Some applications, notably 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. This practical need supports investigations of algorithms for efficient manipulation of large integers.

The conventional pen-and-pencil algorithm (or) method requires n^2 digital multiplication Through divide and conquer technique we can design an algorithm with fewer than n^2 digital multiplications.

To demonstrate the basic idea of the algorithm, let us start with a case of two-digits integers

$$23 \times 14$$

These numbers can be represented as follows

$$23 = 2 \cdot 10^1 + 3 \cdot 10^0$$

$$14 = 1 \cdot 10^1 + 4 \cdot 10^0$$

Now let us multiply them

~~23×14 = (2×1)~~

$$23 \times 14 = (2 \cdot 10^1 + 3 \cdot 10^0) * (1 \cdot 10^1 + 4 \cdot 10^0)$$

$$\boxed{(2*1)10^2 + (2*4 + 3*1)10^1 + (3*4)10^0}$$

The above formula yields the correct answer of 332, but it uses the same four digit multiplications as the pen-and-pencil algorithm.

We can compute the middle term with one digital multiplication by taking advantage of the products $2*1$ and $3*4$ that need to be computed anyway.

$$2*4 + 3*1 = (2+3) * (1+4) - 2*1 - 3*4$$

For any pair of two digit numbers

$a = a_1 a_0$ and $b = b_1 b_0$

and their product c can be computed by the formula

$$c = a*b = c_2 10^2 + c_1 10^1 + c_0$$

where,

$c_2 = a_1 * b_1$ is the product of their first digits.

$c_0 = a_0 * b_0$ is the product of their second digits

$c_1 = (a_1 + a_0) * (b_1 + b_0) - (c_2 + c_0)$ is the product of the sum of the a's digits & the sum of the b's digits minus the sum of c_2 and c_0.

4 digit 1234×5678

$= 12 \times 10^2 + 34 \times 10^0 * 56 \times 10^2 + 78 \times 10^0$

$= \underbrace{(12 \times 56)}_{①} 10^4 + (34 \times 56 + 12 \times 78) 10^2 + \underbrace{(34 \times 78)}_{②} 10^0$

$\Downarrow$

$\underbrace{(12+34) \times (56+78) - (12 \times 56) - (34 \times 78)}_{③}$

2 digit

12×56

$1 \times 10^1 + 2 \times 10^0 \times 5 \times 10^1 + 6 \times 10^0$

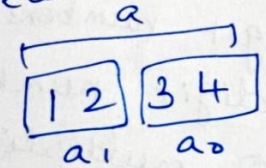
$(1 \times 5) 10^2 + (1 \times 6 + 2 \times 5) 10^1 + (2 \times 6) 10^0$

$\downarrow$

$(1+2) \times (5 \times 6) - (1 * 5) - (2 \times 6)$

Multiplying two n-digit numbers. Let us take a and b are two n digit numbers. We denote the first half of a a's digits by a_1 and second half by a_0

eg



for b_1 the notations are b_1 and b_0 respectively.

In these notations $a = a_1 a_0$ implies that

$a = a_1 10^{n/2} + a_0$ and

$b = b_1 10^{n/2} + b_0$

Now we get

$$c = a * b = (a_1 10^{n/2} + a_0) * (b_1 10^{n/2} + b_0)$$

$$= (a_1 * b_1) 10^n + (a_1 * b_0 + a_0 * b_1) 10^{n/2} + (a_0 * b_0)$$

$$= c_2 10^n + c_1 10^{n/2} + c_0$$

where

$c_2 = a_1 * b_1$ is the product of their first halves.

$c_0 = a_0 * b_0$ is the product of their second halves.

$c_1 = (a_1 + a_0) * (b_1 + b_0) - (c_2 + c_0)$ is the product of the sum of the a's halves and the sum of the b's halves minus the sum of c_2 and c_0.

Since multiplication of n digit numbers requires 3 multiplications of n/2 digit numbers, the recurrence for the number of multiplication $M(n)$ is

$$M(n) = 3M(n/2) \quad \text{for } n > 1, \quad M(1) = 1$$

Strassen's Matrix Multiplication

Matrix multiplication of two 2×2 matrices A and B requires eight multiplication

Eg

$$\begin{bmatrix} 1 & 3 \\ 7 & 8 \end{bmatrix} * \begin{bmatrix} 2 & 3 \\ 1 & 6 \end{bmatrix} = \begin{bmatrix} 1*2+3*1 & 1*3+3*6 \\ 7*2+8*1 & 7*3+8*6 \end{bmatrix}$$

In general, the above can be represented as

$$\begin{bmatrix} a_{00} & a_{01} \\ a_{10} & a_{11} \end{bmatrix} * \begin{bmatrix} b_{00} & b_{01} \\ b_{10} & b_{11} \end{bmatrix} = \begin{bmatrix} a_{00}b_{00}+a_{01}b_{10} & a_{00}b_{01}+a_{01}b_{11} \\ a_{10}b_{00}+a_{11}b_{10} & a_{10}b_{01}+a_{11}b_{11} \end{bmatrix}$$

$$= \begin{bmatrix} C_{00} & C_{01} \\ C_{10} & C_{11} \end{bmatrix}$$

Through divide and conquer, we can find the product of two 2×2 matrices A and B with just seven multiplications. This is accomplished by using the following formula.

Eg

$$\begin{bmatrix} c_{00} & c_{01} \\ c_{10} & c_{11} \end{bmatrix} = \begin{bmatrix} a_{00} & a_{01} \\ a_{10} & a_{11} \end{bmatrix} * \begin{bmatrix} b_{00} & b_{01} \\ b_{10} & b_{11} \end{bmatrix}$$

$$= \begin{bmatrix} m_1 + m_4 - m_5 + m_7 & m_3 + m_5 \\ m_2 + m_4 & m_1 + m_3 - m_2 + m_6 \end{bmatrix}$$

$$m_1 = (a_{00} + a_{11}) * (b_{00} + b_{11})$$

$$m_2 = (a_{10} + a_{11}) * b_{00}$$

$$m_3 = a_{00} * (b_{01} - b_{11})$$

$$m_4 = a_{11} * (b_{10} - b_{00})$$

$$m_5 = (a_{00} + a_{01}) * b_{11}$$

$$m_6 = (a_{10} - a_{00}) * (b_{00} + b_{01})$$

$$m_7 = (a_{01} - a_{11}) * (b_{10} + b_{11})$$

Let A and B be two $n \times n$ matrices where n is a power of 2

We can divide A, B and their product C into four $n/2 \times n/2$ sub matrices.

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

C_{00} can be computed either as

$$A_{00} * B_{00} + A_{01} * B_{10}$$

(or)

$$M_1 + M_4 - M_5 + M_7$$

Where

M_1, M_4, M_5 and M_7 are found by Strassen's formulas, with the numbers replaced by the corresponding submatrices.

Recurrence relation.

$$M(n) = 7M(n/2) \quad \text{for } n > 1$$
$$M(1) = 1$$

$$M(n) = 7^{\log_2 n} = n^{\log_2 7} \approx n^{2.807}$$

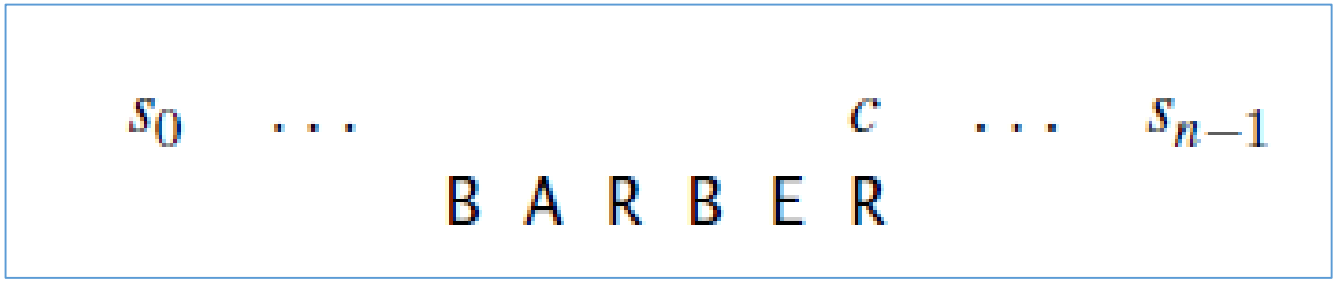
Input Enhancement in String Matching

Input Enhancement in String Matching

- The technique of input enhancement can be applied to the problem of string matching
- Several faster algorithms have been discovered. Most of them exploit the input-enhancement idea
- **Preprocess the pattern to get some information about it, store this information** in a table, and then use this information during an actual search for the pattern in a given text

Horspool's Algorithm

Searching for the pattern BARBER in some text



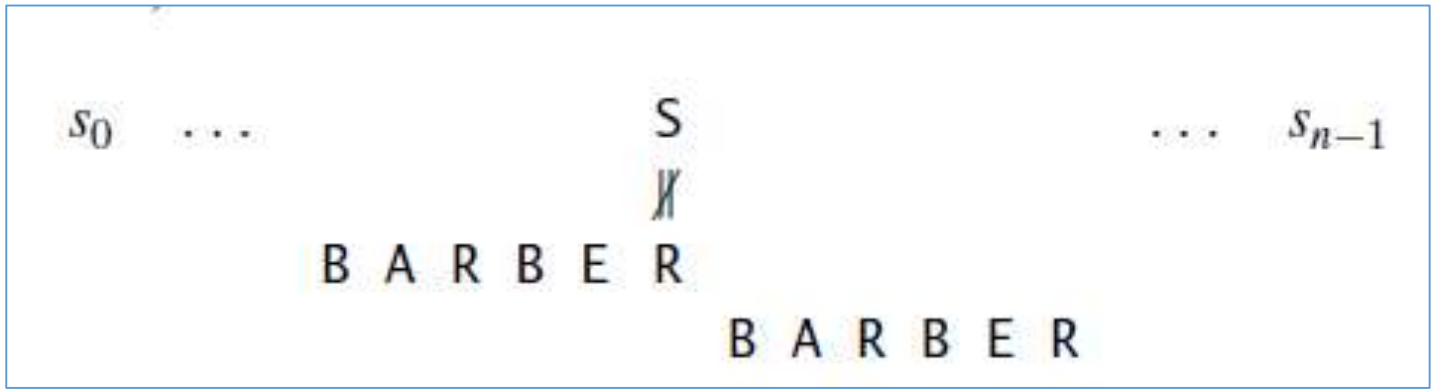
i. Starting with the last R of the pattern and moving right to left, we compare the corresponding pairs of characters in the pattern and the text. If all the pattern's characters match successfully, a matching substring is found

ii. If a mismatch occurs, we need to shift the pattern to the right. Clearly, we would like to make as large a shift as possible without risking the possibility of missing a matching substring in the text.

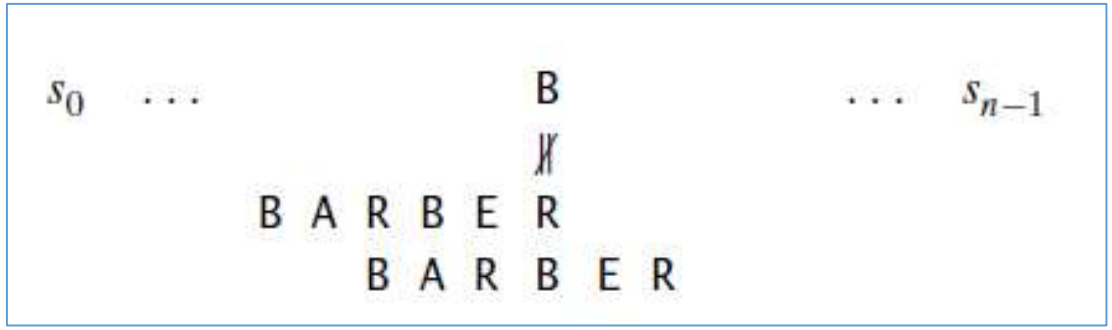
- Horspool's algorithm determines the size of such a shift by looking at the character *c* of the text that is aligned against the last character of the pattern.

In general, the following four possibilities can occur.

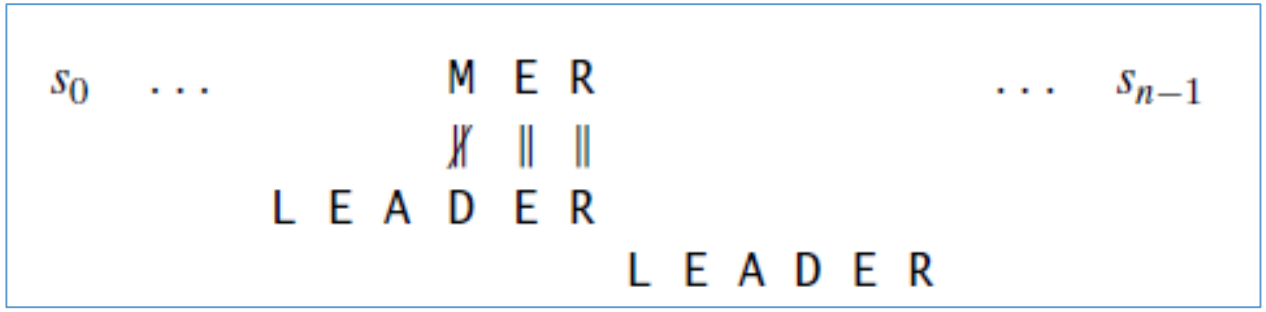
Case 1 If there are no *c*'s in the pattern—e.g., *c* is letter S in our example—we can safely shift the pattern by its entire length



Case 2 If there are **occurrences of character *c* in the pattern** but it is not the last one there—e.g., *c* is letter B in our example—the shift should align the rightmost occurrence of *c* in the pattern with the *c* in the text:



Case 3 If *c* happens to be the last character in the pattern but there are no *c*'s among its other *m* − 1 characters—e.g., *c* is letter R in our example—the situation is similar to that of Case 1 and the pattern should be shifted by the entire pattern's length *m*:



Case 4 Finally, if c happens to be the last character in the pattern and there are other c's among its first $m - 1$ characters—e.g., c is letter R in our example—the situation is similar to that of Case 2 and the rightmost occurrence of c among the first $m - 1$ characters in the pattern should be aligned with the text's c:

Horspool's Algorithm -Example

Shift Table

character c	A	B	C	D	E	F	. . .	R	. . .	Z	_
shift $t(c)$	4	2	6	6	1	6	6	3	6	6	6

```
J I M _ S A W _ M E _ I N _ A _ B A R B E R S H O P
B A R B E R             B A R B E R
          B A R B E R           B A R B E R
              B A R B E R               B A R B E R
```

Horspool's algorithm

Step 1 For a given pattern of length m and the alphabet used in both the pattern and text, construct the shift table as described above.

Step 2 Align the pattern against the beginning of the text.

Step 3 Repeat the following until either a matching substring is found or the pattern reaches beyond the last character of the text. Starting with the last character in the pattern, compare the corresponding characters in the pattern and text until either all m characters are matched (then stop) or a mismatching pair is encountered. In the latter case, retrieve the entry $t(c)$ from the c's column of the shift table where c is the text's character currently aligned against the last character of the pattern, and shift the pattern by $t(c)$ characters to the right along the text.

ALGORITHM *ShiftTable*$(P[0..m-1])$

//Fills the shift table used by Horspool's and Boyer-Moore algorithms
//Input: Pattern $P[0..m-1]$ and an alphabet of possible characters
//Output: $Table[0..size-1]$ indexed by the alphabet's characters and
// filled with shift sizes computed by formula (7.1)
for $i \leftarrow 0$ **to** $size-1$ **do** $Table[i] \leftarrow m$
for $j \leftarrow 0$ **to** $m-2$ **do** $Table[P[j]] \leftarrow m-1-j$
return *Table*

ALGORITHM *HorspoolMatching*$(P[0..m-1], T[0..n-1])$

```
//Implements Horspool's algorithm for string matching
//Input: Pattern P[0..m − 1] and text T[0..n − 1]
//Output: The index of the left end of the first matching substring
//          or −1 if there are no matches
ShiftTable(P[0..m − 1])          //generate Table of shifts
i ← m − 1                        //position of the pattern's right end
while i ≤ n − 1 do
    k ← 0                        //number of matched characters
    while k ≤ m − 1 and P[m − 1 − k] = T[i − k] do
        k ← k + 1
    if k = m
        return i − m + 1
    else i ← i + Table[T[i]]
return −1
```

Boyer – Moore Algorithms

- The algorithm compares the pattern P with the substring of sequence T within a sliding window in the right-to-left pattern
- The bad character rule and good suffix rule are used to determine the movement of sliding window

Boyer – Moore Algorithms

Based on same two ideas:

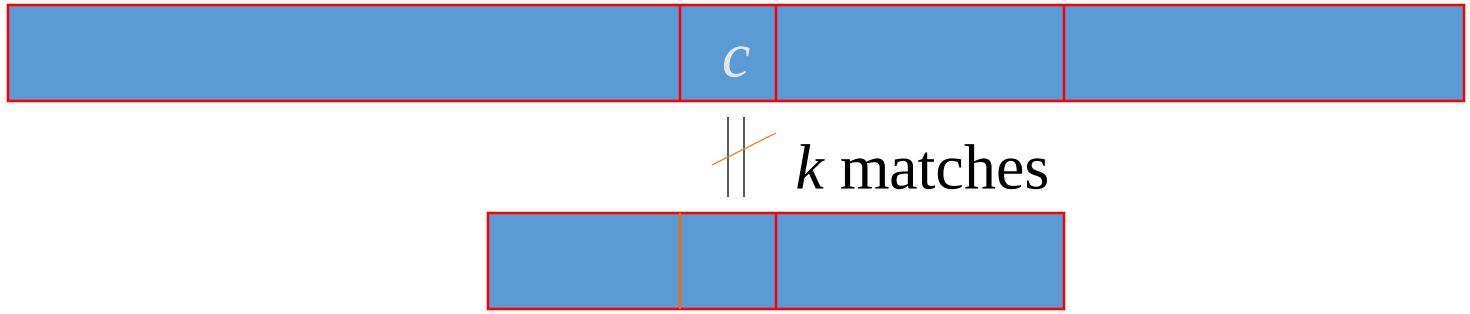
- 1. Comparing pattern characters to text from *right to left*
- 2. Precomputing shift sizes, but now with *two* tables:
  1. *bad-symbol table* = Horspool's table. But in use,
     a. shift uses text character at mismatch (not last)
     b. Shift uses *length* of matching suffix
  2. *good-suffix table* shift based on matched suffix itself

Bad-symbol shift in Boyer-Moore algorithm

- If the rightmost character of the pattern doesn't match, BM algorithm acts as Horspool's
- If the rightmost character of the pattern does match, BM compares preceding characters right to left until either all pattern's characters match or a mismatch on text's character c is encountered after $k > 0$ matches

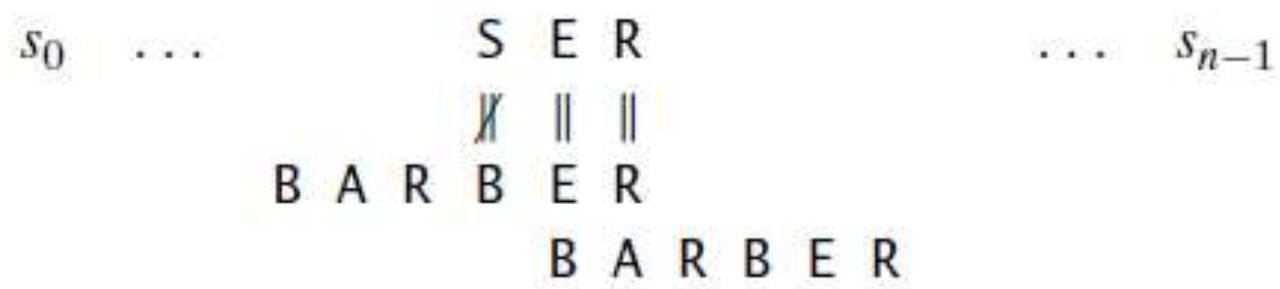
text

pattern

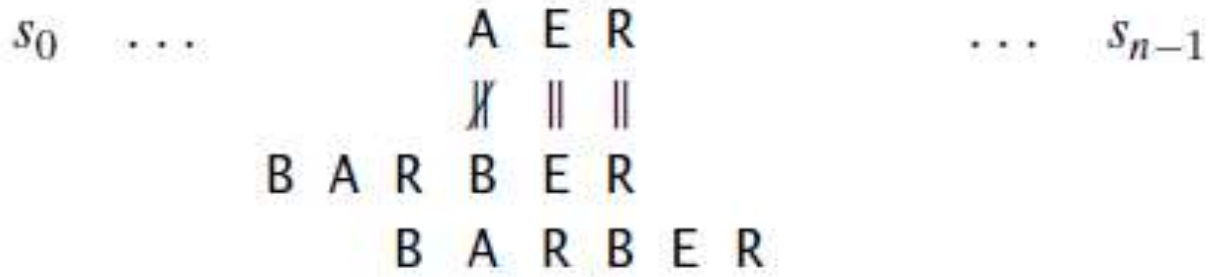


bad-symbol shift $d_1 = \max\{t_1(c) - k, 1\}$

For example, if we search for the pattern BARBER in some text and match the last two characters before failing on letter S in the text, we can shift the pattern by $t_1(\mathsf{S}) - 2 = 6 - 2 = 4$ positions:



The same formula can also be used when the mismatching character c of the text occurs in the pattern, provided $t_1(c) - k > 0$. For example, if we search for the pattern BARBER in some text and match the last two characters before failing on letter A, we can shift the pattern by $t_1(\mathsf{A}) - 2 = 4 - 2 = 2$ positions:



Good-suffix shift

- Good-suffix shift d_2 is applied after $0 < k < m$ characters were matched
- $d_2(k)$ = the distance between matched suffix of size k and its rightmost occurrence in the pattern that is not preceded by the same character as the suffix

  Example: CABABA $d_2(1) = 4$

- If there is no such occurrence, match the longest part of the k-character suffix with corresponding prefix;
  if there are no such suffix-prefix matches, $d_2(k) = m$

  Example: WOWWOW: $d_2(2) = 5$, $d_2(3) = 3$, $d_2(4) = 3$, $d_2(5) = 3$

Good-suffix shift in Boyer-Moore algorithm

- Good-suffix shift d_2 is applied after $0 < k < m$ last characters were matched
- $d_2(k)$ = the distance between matched suffix of size k and its rightmost occurrence in the pattern that is not preceded by the same character as the suffix

k	pattern	d_2
1	ABCBAB	2
2	ABCBAB	4
3	ABCBAB	4
4	ABCBAB	4
5	ABCBAB	4

The Boyer-Moore algorithm

Step 1 For a given pattern and the alphabet used in both the pattern and the text, construct the bad-symbol shift table as described earlier.

Step 2 Using the pattern, construct the good-suffix shift table as described earlier.

Step 3 Align the pattern against the beginning of the text.

Step 4 Repeat the following step until either a matching substring is found or the pattern reaches beyond the last character of the text. Starting with the last character in the pattern, compare the corresponding characters in the pattern and the text until either all m character pairs are matched (then stop) or a mismatching pair is encountered after $k \geq 0$ character pairs are matched successfully. In the latter case, retrieve the entry $t_1(c)$ from the c's column of the bad-symbol table where c is the text's mismatched character. If $k > 0$, also retrieve the corresponding d_2 entry from the good-suffix table. Shift the pattern to the right by the

Boyer - Moore Algorithm

number of positions computed by the formula

$$d = \begin{cases} d_1 & \text{if } k = 0, \\ \max\{d_1, d_2\} & \text{if } k > 0, \end{cases}$$

where $d_1 = \max\{t_1(c) - k, 1\}$.

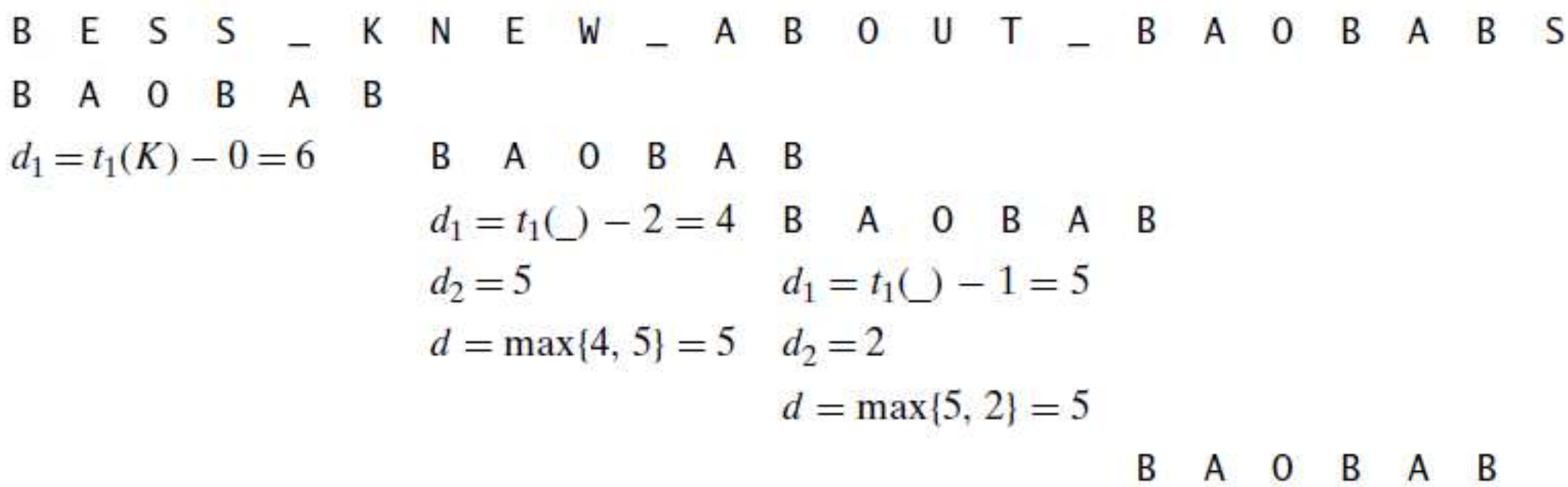


FIGURE 7.3 Example of string matching with the Boyer-Moore algorithm.

Example of Boyer-Moore alg. application

A	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	_
1	2	6	6	6	6	6	6	6	6	6	6	6	6	3	6	6	6	6	6	6	6	6	6	6	6	6

B E S S _ K N E W _ A B O U T _ B A O B A B S

B A O B A B

$d_1 = t_1(\text{K}) = 6$

B A O B A B

$d_1 = t_1(\_)\text{-}2 = 4$

$d_2(2) = 5$

B A O B A B

$d_1 = t_1(\_)\text{-}1 = 5$

$d_2(1) = 2$

B A O B A B (success)

k	pattern	d_2
1	BAO**BAB**	2
2	**B**AOB**AB**	5
3	**BA**O**BAB**	5
4	**BAOBAB**	5
5	**BAOBAB**	5

Exercise

- Build the good suffix table for pattern WOWWOW

k	pattern	d_2
1	WOW**W**OW	2
2	**W**OWWO*W*	5
3	**WOW**WOW	3
4	**WOW**WOW	3
5	**WOW**WOW	3

Case 1

Case 2: longest part of OW with a matched prefix is W

Case 1

Case 2: longest part of WWOW with a matched prefix is WOW

Case 2: longest part of WWOW with a matched prefix is WOW

Exercise

W H I C H _ F I N A L L Y _ H A L T S _ _ A T _T H A T

Find pattern A T_ T H A T in the above text

Transform-and-Conquer

Transform-and-Conquer

- This technique is called as ***transform-and-conquer*** because these methods work as two-stage procedures.
- First, in the **transformation stage**, the problem's instance is modified to be, for one reason or another, more amenable to solution.
- Then, in the second or **conquering stage**, it is solved.

Transform-and-Conquer (Variations)

There are **three major variations** of this idea that differ by what we transform a given instance to

- Transformation to a simpler or more convenient instance of the same problem—we call it instance simplification (Pre-Sorting)
- Transformation to a different representation of the same instance—we call it ***representation change*** (heaps and heapsort)
- Transformation to an instance of a different problem for which an algorithm is already available—we call it ***problem reduction***.

Question

Identify the variation of Transform and Conquer technique for the following scenarios. Justify your answer.

Scenario-1	Scenario-2	Scenario -3
Finding LCM using GCD. (Assume GCD algorithm already exist)	Search for a given K in A[0..n-1] Stage 1: Sort the array by an efficient sorting algorithm Stage 2: Apply binary search	Sorting a set of elements in ascending order by Heap Sort Technique by constructing heap data structure.

Presorting

- Presorting is an example for instance simplification.
- Many questions about lists are easier to answer if the lists are sorted.
- Time efficiency of the problem's algorithm is dominated by the algorithm used for sorting the list.

Problem of checking element uniqueness in an array

Limitation of brute-force algorithm

Compares pairs of the array's elements until either two equal elements are found or no more pairs were left. Worst case efficiency = $\Theta(n^2)$

Using presorting

Presorting helps to 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

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

$$T(n) = T_{sort}(n) + T_{scan}(n) \in \Theta(n \log n) + \Theta(n) = \Theta(n \log n).$$

Question

- For the given array, write an algorithm to determine array uniqueness using the concept of presorting and analyze its time complexity.

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.

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

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

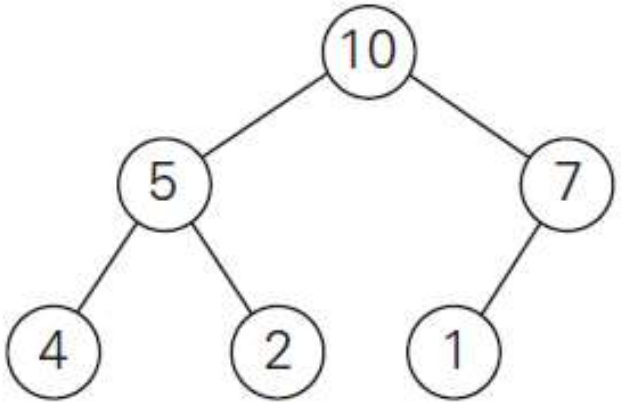
```
//Computes the mode of an array by sorting it first
//Input: An array A[0..n − 1] of orderable elements
//Output: The array's mode
sort the array A
i ← 0                              //current run begins at position i
modefrequency ← 0                  //highest frequency seen so far
while i ≤ n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

Notion of the Heap

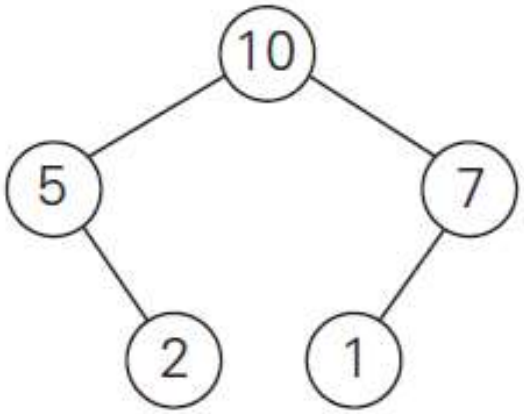
A ***heap*** can be defined as a binary tree with keys assigned to its nodes, one key per node, provided the following two conditions are met

- The shape property—the binary tree is essentially complete (or simply complete), i.e., all its levels are full except possibly the last level, where only some rightmost leaves may be missing.

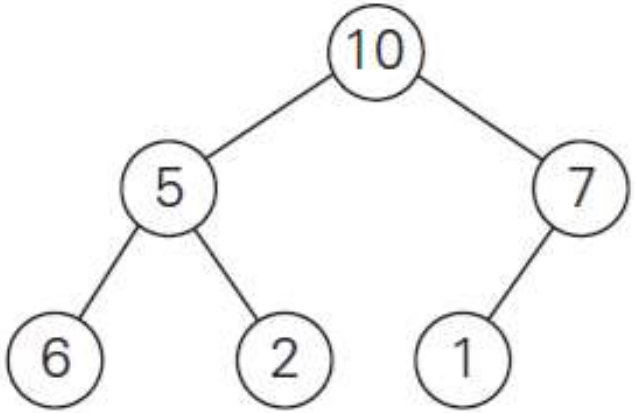
- The parental dominance or heap property—the key in each node is greater than or equal to the keys in its children



Heap

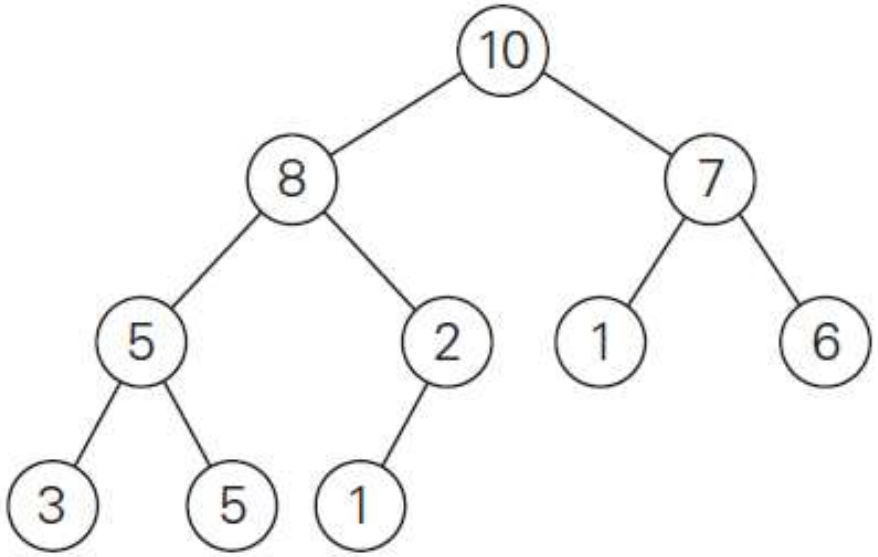


Not Heap

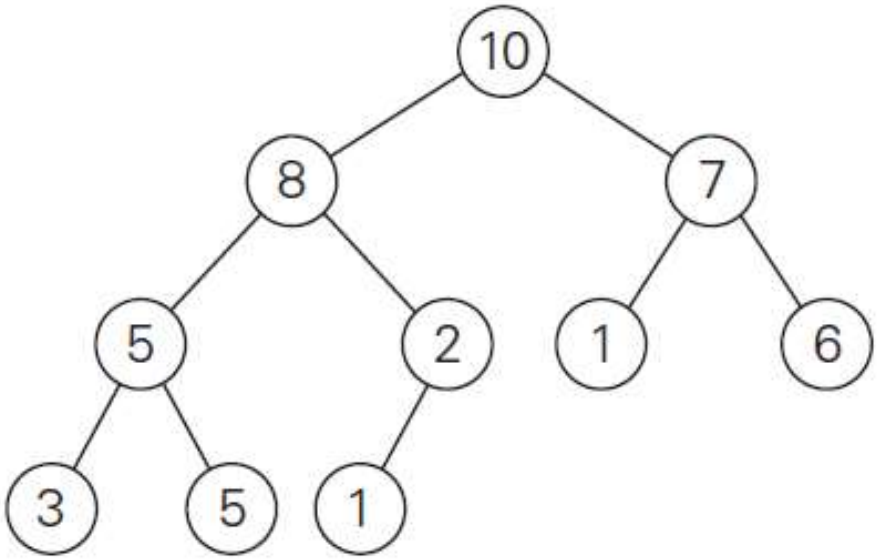


Not Heap

- The key values in a heap are ordered top down
- There is no left-to-right order in key values
- There is no relationship among key values for nodes either on the same level of the tree



Heap and its array representation



the array representation

index	0	1	2	3	4	5	6	7	8	9	10
value		10	8	7	5	2	1	6	3	5	1

parents | leaves

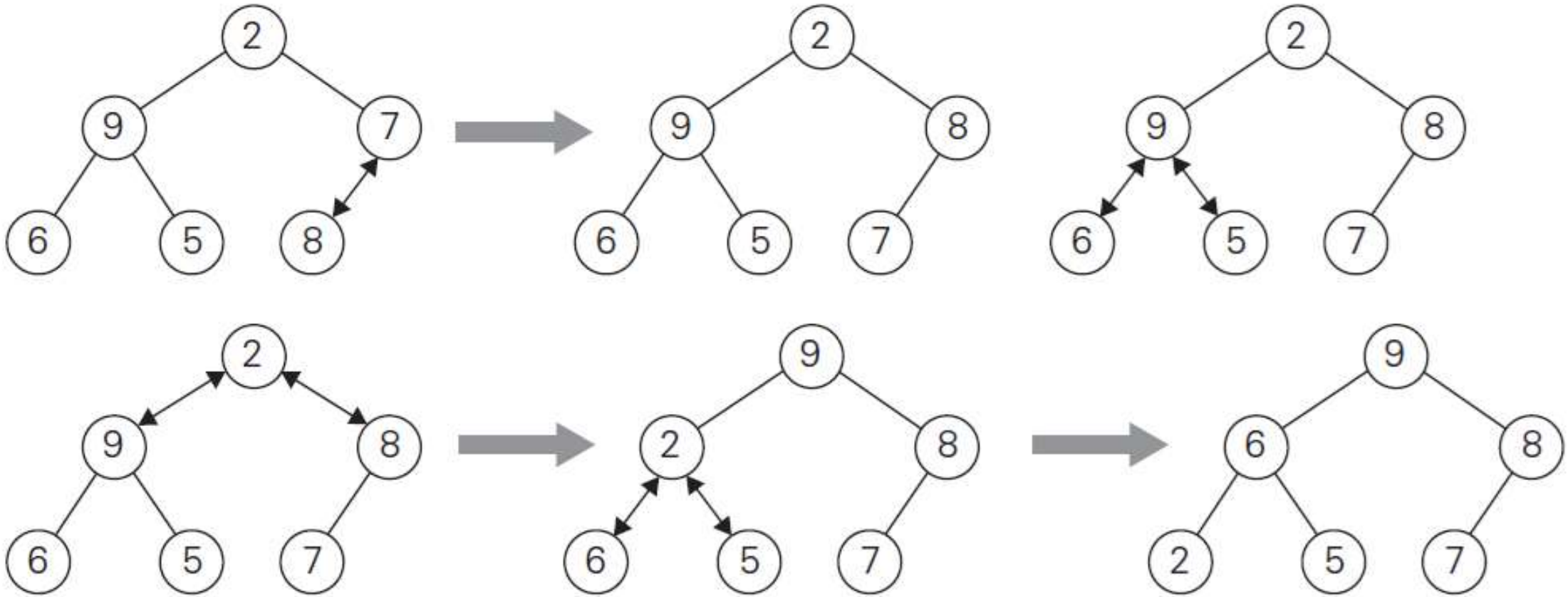
Properties of Heaps

1. There exists exactly one essentially complete binary tree with n nodes. Its height is equal to $\lfloor \log_2 n \rfloor$.
2. The root of a heap always contains its largest element.
3. A node of a heap considered with all its descendants is also a heap.
4. A heap can be implemented as an array by recording its elements in the top-down, left-to-right fashion. It is convenient to store the heap's elements in positions 1 through n of such an array, leaving $H[0]$ either unused or putting there a sentinel whose value is greater than every element in the heap. In such a representation,
   a. the parental node keys will be in the first $\lfloor n/2 \rfloor$ positions of the array, while the leaf keys will occupy the last $\lceil n/2 \rceil$ positions;
   b. the children of a key in the array's parental position i $(1 \le i \le \lfloor n/2 \rfloor)$ will be in positions $2i$ and $2i + 1$, and, correspondingly, the parent of a key in position i $(2 \le i \le n)$ will be in position $\lfloor i/2 \rfloor$.

Construct a heap - ***bottom-up heap construction***

- It initializes the essentially **complete binary tree with *n* nodes by placing keys in the order given** and then “heapifies” the tree as follow

  - Starting with the last parental node, the algorithm checks whether the parental dominance holds for the key in this node.
  - If it does not, the algorithm exchanges the node’s key K with the larger key of its children and checks whether the parental dominance holds for K in its new position
  - This process continues until the parental dominance for *K* is satisfied.
  - After completing the “heapification” of the subtree rooted at the current parental node, the algorithm proceeds to do the same for the node’s immediate predecessor. The algorithm stops after this is done for the root of the tree.



Bottom-up construction of a heap for the list 2, 9, 7, 6, 5, 8. The double-headed arrows show key comparisons verifying the parental dominance.

ALGORITHM *HeapBottomUp*$(H[1..n])$

```
//Constructs a heap from elements of a given array
// by the bottom-up algorithm
//Input: An array H[1..n] of orderable items
//Output: A heap H[1..n]
for i ← ⌊n/2⌋ downto 1 do
    k ← i;   v ← H[k]
    heap ← false
    while not heap and 2 * k ≤ n do
        j ← 2 * k
        if j < n    //there are two children
            if H[j] < H[j + 1]  j ← j + 1
        if v ≥ H[j]
            heap ← true
        else H[k] ← H[j];    k ← j
    H[k] ← v
```

Maximum Key Deletion from a heap

Step 1 Exchange the root's key with the last key K of the heap.

Step 2 Decrease the heap's size by 1.

Step 3 "Heapify" the smaller tree by sifting K down the tree exactly in the same way we did it in the bottom-up heap construction algorithm. That is, verify the parental dominance for K: if it holds, we are done; if not, swap K with the larger of its children and repeat this operation until the parental dominance condition holds for K in its new position.

Heap Sort Algorithm

Stage 1 (heap construction): Construct a heap for a given array.

Stage 2 (maximum deletions): Apply the root-deletion operation $n - 1$ times to the remaining heap.

Stage 1 (heap construction)

2 9 **7** 6 5 8

2 **9** 8 6 5 7

2 9 8 6 5 7

9 **2** 8 6 5 7

9 6 8 2 5 7

Stage 2 (maximum deletions)

9 6 8 2 5 7

7 6 8 2 5 | **9**

8 6 7 2 5

5 6 7 2 | **8**

7 6 5 2

2 6 5 | **7**

6 2 5

5 2 | **6**

5 2

2 | **5**

2

Question

- Create a max heap tree for the following list of elements. Show step-by- step construction of tree and sorting of the elements. Also, design an algorithm for the same.
- {55, 28, 35, 78, 110, 48, 88}

Horner’s Rule

- Horner’s method is an algorithm **for polynomial evaluation**
- It allows the evaluation of a polynomial of degree n with only n multiplications and additions
- Horner’s rule is a good example of the representation-change technique
- Horner’s method can be used to evaluate polynomial in O(n) time

Horner's Rule

Polynomial Equation

$$p(x) = a_n x^n + a_{n-1} x^{n-1} + \cdots + a_1 x + a_0$$

The following formula is obtained by successively taking x as a common factor in the remaining polynomials of diminishing degrees:

$$p(x) = (\cdots (a_n x + a_{n-1})x + \cdots)x + a_0.$$

$$
\begin{aligned}
p(x) &= 2x^4 - x^3 + 3x^2 + x - 5 \\
&= x(2x^3 - x^2 + 3x + 1) - 5 \\
&= x(x(2x^2 - x + 3) + 1) - 5 \\
&= x(x(x(2x - 1) + 3) + 1) - 5.
\end{aligned}
$$

Evaluate $p(x) = 2x^4 - x^3 + 3x^2 + x - 5$ at $x = 3$.

coefficients	2	-1	3	1	-5
$x = 3$	2	$3 \cdot 2 + (-1) = 5$	$3 \cdot 5 + 3 = 18$	$3 \cdot 18 + 1 = 55$	$3 \cdot 55 + (-5) = 160$

ALGORITHM *Horner*$(P[0..n], x)$

//Evaluates a polynomial at a given point by Horner's rule
//Input: An array $P[0..n]$ of coefficients of a polynomial of degree n,
// stored from the lowest to the highest and a number x
//Output: The value of the polynomial at x
$p \leftarrow P[n]$
for $i \leftarrow n - 1$ **downto** 0 **do**
    $p \leftarrow x * p + P[i]$
return p