Characteristics of Algorithm

Input: Zero or more quantities are externally supplied

Definiteness: Each instruction is clear and unambiguous

Finiteness: The algorithm terminates in a finite number of steps.

Effectiveness: Each instruction must be primitive and feasible

Output: At least one quantity is produced

Algorithm

Why do we need Algorithms?

- It is a tool for solving well-specified Computational Problem.
- Problem statement specifies in general terms relation between input and output
- Algorithm describes computational procedure for achieving input/output relationship
  This Procedure is irrespective of implementation details

Why do we need to study algorithms?

Exposure to different algorithms for solving various problems helps develop skills to design algorithms for the problems for which there are no published algorithms to solve it

Algorithm Design Technique

What do you mean by Algorithm Design Techniques?

General Approach to solving problems algorithmically .
Applicable to a variety of problems from different areas of computing

Various Algorithm Design Techniques

- Brute Force
- Divide and Conquer
- Decrease and Conquer
- Transform and Conquer
- Dynamic Programming
- Greedy Technique
- Branch and Bound
- Backtracking

Importance Framework for designing and analyzing algorithms for new problems

Methods of Specifying an Algorithm

- **Natural language**
  - Ambiguous
- **Pseudocode**
  - A mixture of a natural language and programming language-like structures
  - Precise and succinct.
  - Pseudocode in this course
    - omits declarations of variables
    - use indentation to show the scope of such statements as for, if, and while.
    - use ← for assignment
- **Flowchart**
  - Method of expressing algorithm by collection of connected geometric shapes

Euclid's Algorithm

Problem: Find gcd(m,n), the greatest common divisor of two nonnegative, not both zero integers m and n

Examples: gcd(60,24) = 12, gcd(60,0) = 60, gcd(0,0) = ?

Euclid's algorithm is based on repeated application of equality

gcd(m,n) = gcd(n, m mod n)

until the second number becomes 0, which makes the problem trivial.

Example: gcd(60,24) = gcd(24,12) = gcd(12,0) = 12

Methods of Specifying an Algorithm

Two descriptions of Euclid's algorithm

Euclid's algorithm for computing gcd(m,n)

Step 1 If n = 0, return m and stop; otherwise go to Step 2

Step 2 Divide m by n and assign the value of the remainder to r

Step 3 Assign the value of n to m and the value of r to n. Go to step 1.

ALGORITHM Euclid(m,n)

//computes gcd(m,n) by Euclid's method

//Input: Two nonnegative,not both zero integers

//Output:Greatest common divisor of m and n

while n ≠ 0 do

    r ← m mod n

    m← n

    n ← r

return m

Algorithm for Sequential Search

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

```
//Searches for a given value in a given array by sequential search
//Input: An array A[0..n − 1] and a search key K
//Output: The index of the first element of A that matches K
//        or −1 if there are no matching elements
i ← 0
while i < n and A[i] ≠ K do
    i ← i + 1
if i < n return i
else return −1
```

Need of Analysis

- To determine resource consumption
  - CPU time
  - Memory space
- Compare different methods for solving the same problem before actually implementing them and running the programs.
- To find an efficient algorithm for solving the problem

Complexity of an algorithm

- A measure of the performance of an algorithm

- An algorithm's performance is characterized by

  - Time complexity

    How fast an algorithm maps input to output as a function of input

  - Space complexity

    amount of memory units required by the algorithm in addition to the memory needed for its input and output

PES UNIVERSITY

How to determine complexity of an algorithm?

- Experimental study(Performance Measurement)
- Theoretical Analysis (Performance Analysis)

Limitations of Performance Measurement

- It is necessary to implement the algorithm, which may be difficult
- Results may not be indicative of the running time on other inputs not included in the experiment.
- In order to compare two algorithms, the same hardware and software environments must be used
- Experimental data though important is not sufficient

Performance Analysis

- Uses a high-level description of the algorithm instead of an implementation
- Characterizes running time as a function of the input size, n.
- Takes into account all possible inputs
- Allows us to evaluate the speed of an algorithm independent of the hardware/software environment

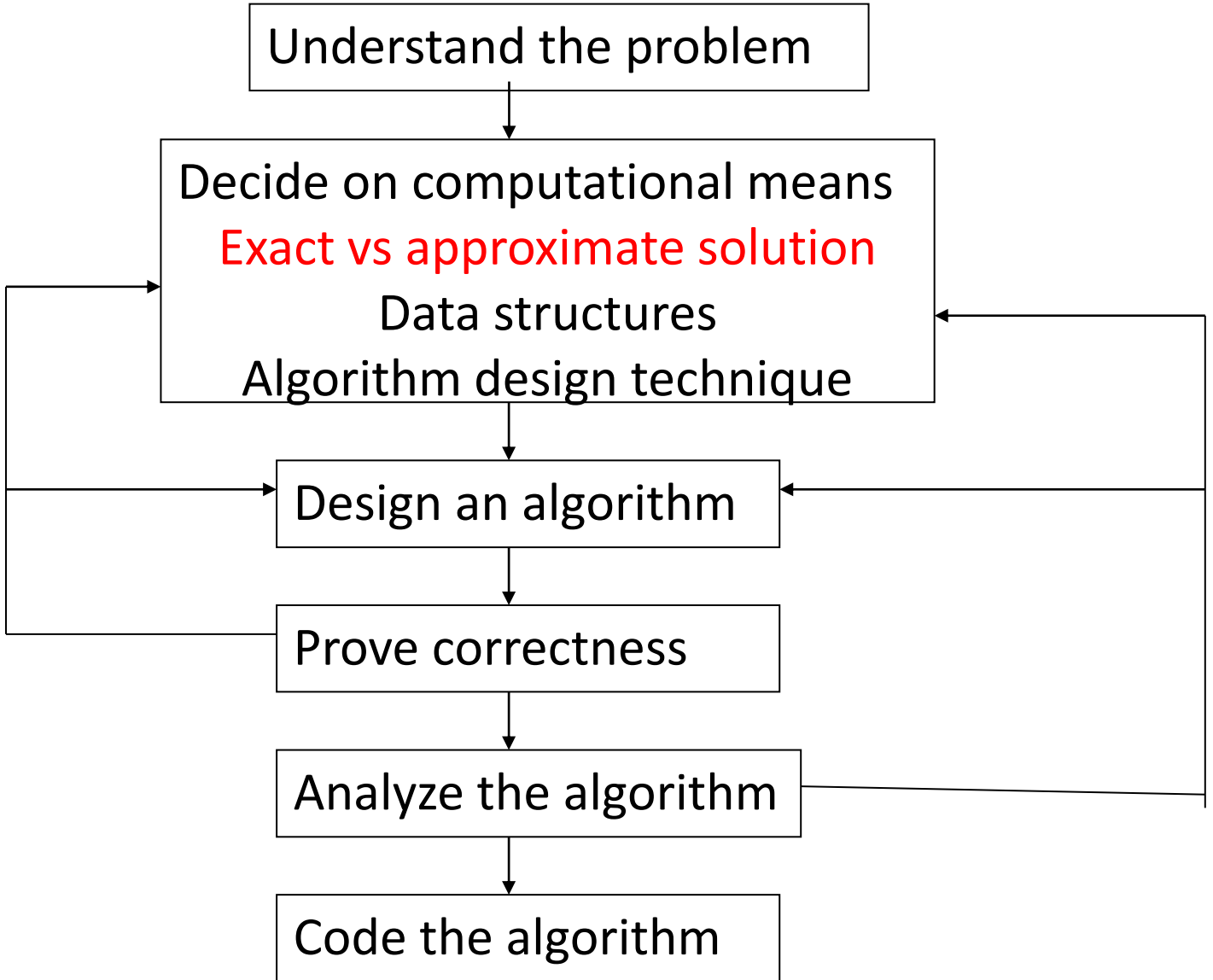
Computational Means

Computational Device the algorithm is intended for

RAM Sequential Algorithms

PRAM Parallel Algorithms

Algorithm Design and Analysis Process



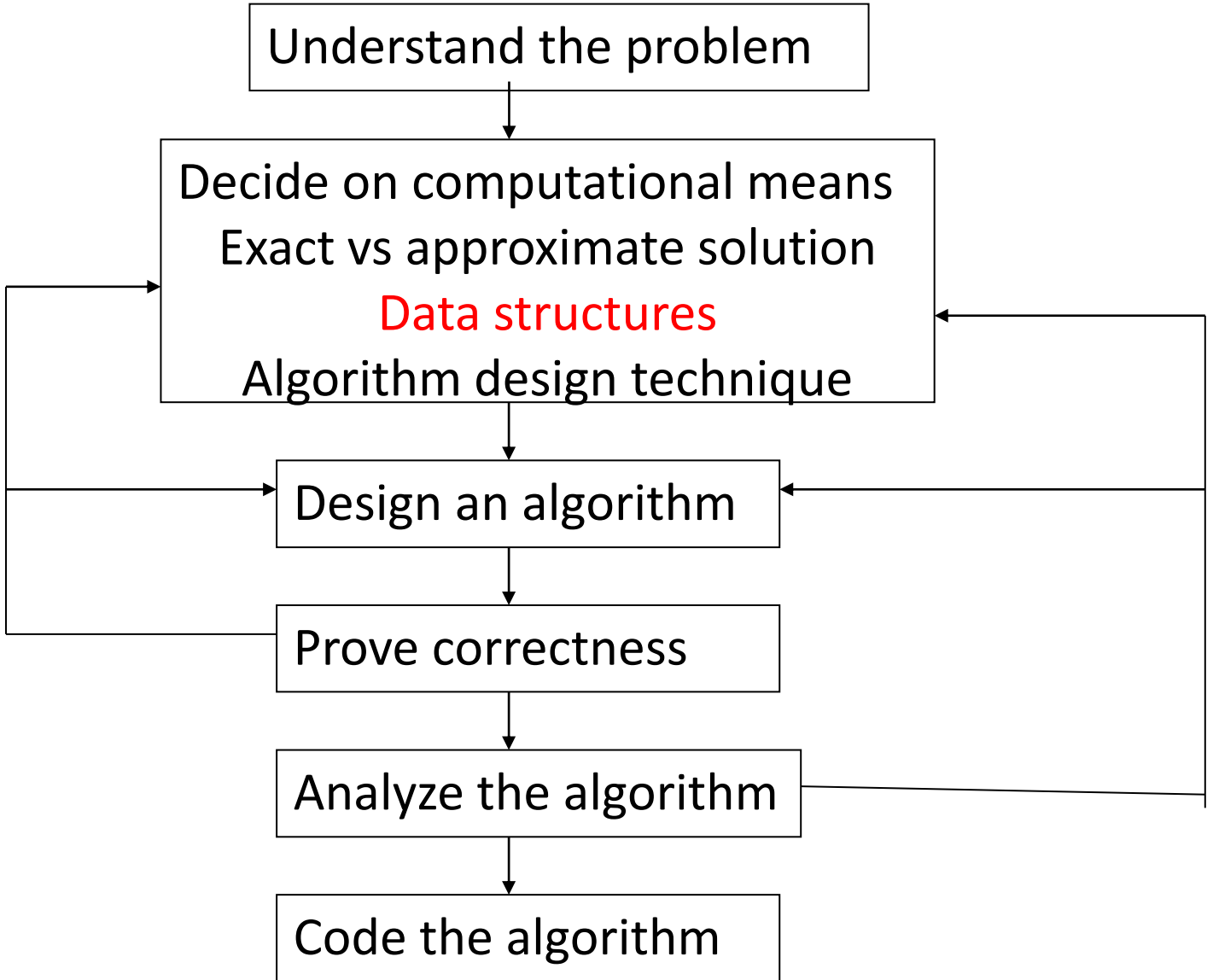
Exact vs approximate solution

Travelling Salesman Problem

NP complete!!!

Approximate algorithm can be used to solve it

Algorithm Design and Analysis Process



Deciding on Data Structure

- Linear

  Linear list, Stack, Queues
- Non Linear

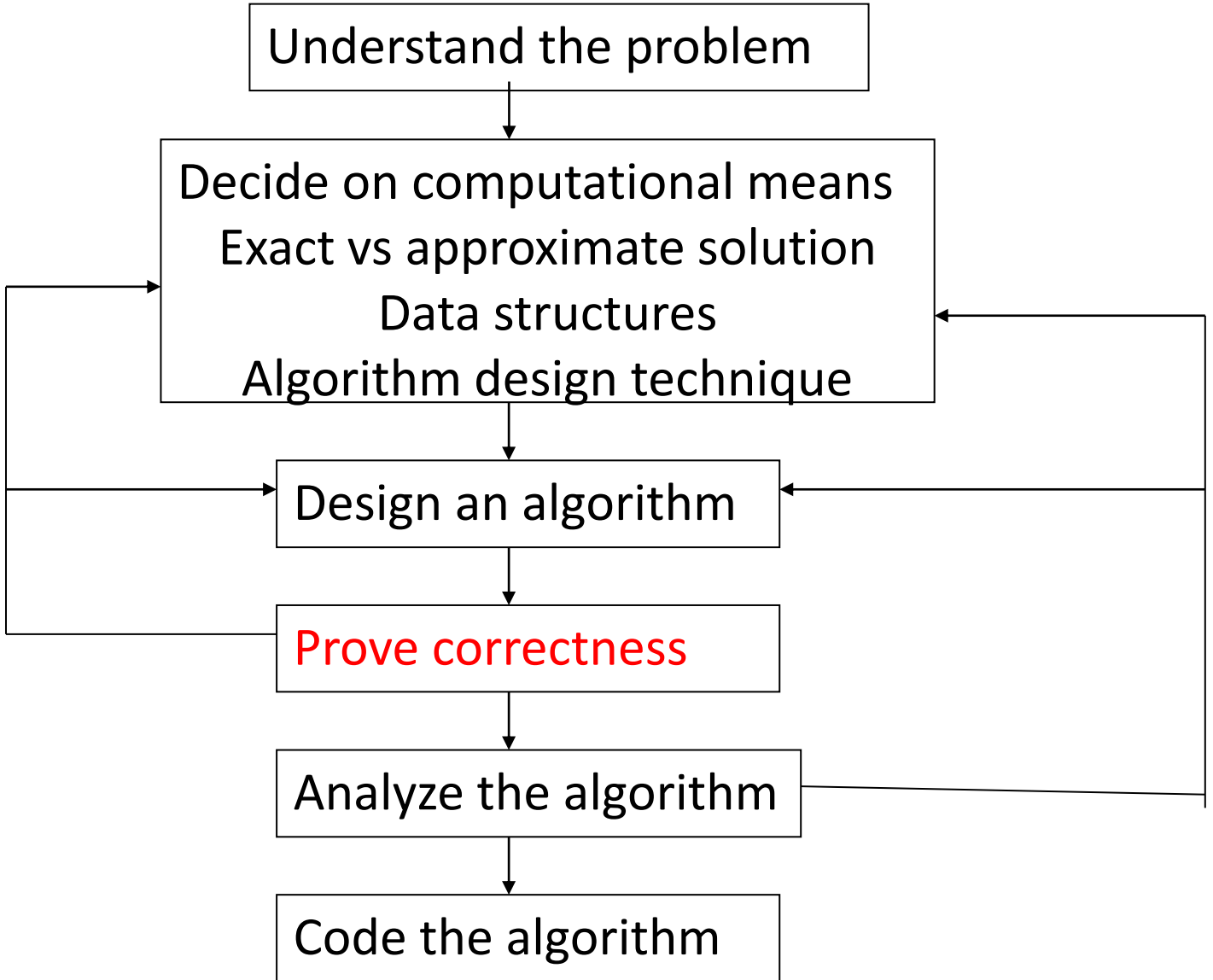
  Trees, Graphs

Choice of Data structure for solving a problem using an algorithm may dramatically impact its time complexity

Dijkstra Algorithm

O(VlogV+E) with Fibonacci heap

Algorithm Design and Analysis Process



Proving Correctness

Exact algorithms

Proving that algorithm yields a correct result for legitimate input in finite amount of time

Approximation algorithms

Error produced by algorithm does not exceed a predefined limit

Analyzing an algorithm

- Efficiency
  - Time efficiency
  - Space efficiency
- Simplicity
- Generality
  - Design an algorithm for the problem posed in more general terms
  - Design an algorithm that can handle a range of inputs that is natural for the problem at hand

Coding an algorithm

- Efficient implementation
- Correctness of program
  - Mathematical Approach: Formal verification for small programs
  - Practical Methods: Testing and Debugging
- Code optimization

Important Problem Types: Sorting

- Rearrange the items of a given list in ascending order.
  - Input: A sequence of n numbers <a1, a2, …, an>
  - Output: A reordering <a'_1, a'_2, …, a'_n> of the input sequence such that $a'_1 \leq a'_2 \leq \ldots \leq a'_n$.
- Why sorting?
  - Help searching
  - Algorithms often use sorting as a key subroutine.
- Sorting key

  A specially chosen piece of information used to guide sorting.

  Example: sort student records by SRN.

Important Problem Types: Sorting

- Rearrange the items of a given list in ascending order.
- Examples of sorting algorithms
  - Selection sort
  - Bubble sort
  - Insertion sort
  - Merge sort
  - Heap sort …
- Evaluate sorting algorithm complexity: the number of key comparisons.
- Two properties
  - Stability: A sorting algorithm is called stable if it preserves the relative order of any two equal elements in its input.
  - In place : A sorting algorithm is in place if it does not require extra memory, except, possibly for a few memory units.

Important Problem Types: Searching

Find a given value, called a search key, in a given set.

Examples of searching algorithms

- Sequential searching
- Binary searching…

Important Problem Types: String Processing

A string is a sequence of characters from an alphabet.

Text strings: letters, numbers, and special characters.

String matching: searching for a given word/pattern in a text.

Text: I am a computer science graduate

Pattern: computer

Important Problem Types: Graph Problems

Definition

Graph G is represented as a pair G= (V, E),
where V is a finite set of vertices and E is a finite set of edges

Modeling real-life problems

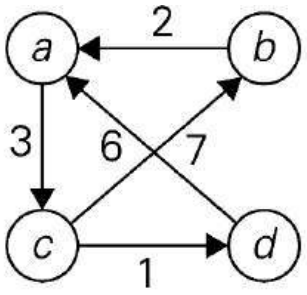
- Modeling WWW
- communication networks
- Project scheduling …

Examples of graph algorithms

- Graph traversal algorithms
- Shortest-path algorithms
- Topological sorting

Shortest paths in a graph

To find the distances from each vertex to all other vertices.



(a)

$$W = \begin{array}{c} \\ a \\ b \\ c \\ d \end{array} \begin{array}{c} \begin{array}{cccc} a & b & c & d \end{array} \\ \begin{bmatrix} 0 & \infty & 3 & \infty \\ 2 & 0 & \infty & \infty \\ \infty & 7 & 0 & 1 \\ 6 & \infty & \infty & 0 \end{bmatrix} \end{array}$$

(b)

$$D = \begin{array}{c} \\ a \\ b \\ c \\ d \end{array} \begin{array}{c} \begin{array}{cccc} a & b & c & d \end{array} \\ \begin{bmatrix} 0 & 10 & 3 & 4 \\ 2 & 0 & 5 & 6 \\ 7 & 7 & 0 & 1 \\ 6 & 16 & 9 & 0 \end{bmatrix} \end{array}$$

(c)

FIGURE 8.5 (a) Digraph. (b) Its weight matrix. (c) Its distance matrix.

Minimum cost spanning tree

- A spanning tree of a connected graph is its connected acyclic sub graph (i.e. a tree).

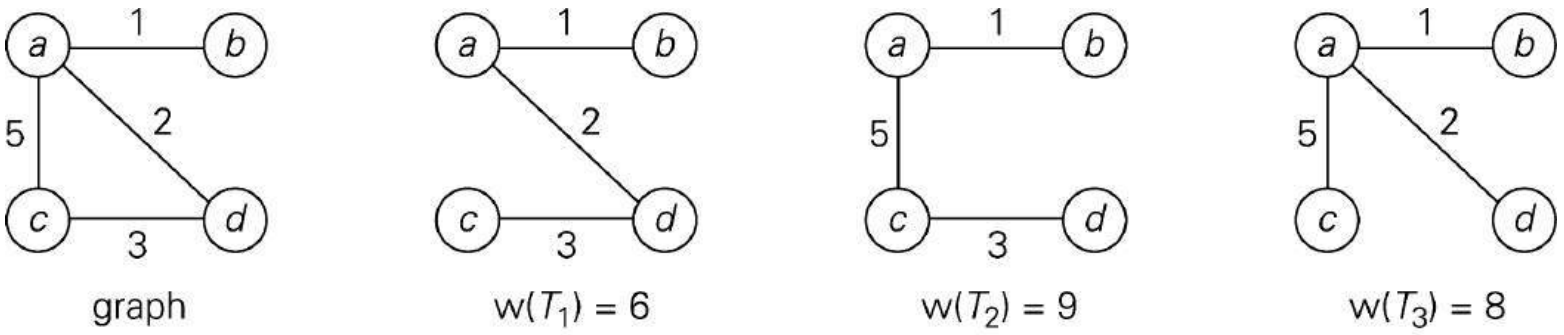
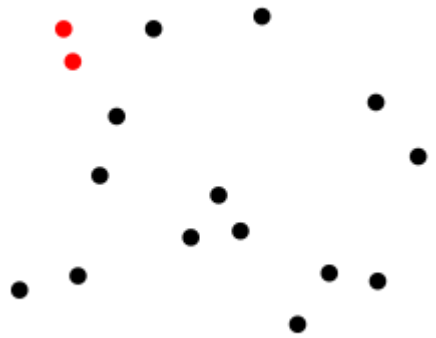


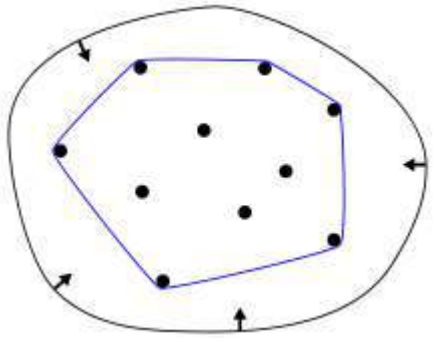
FIGURE 9.1 Graph and its spanning trees; T_1 is the minimum spanning tree

Important Problem Types: Geometric Problems

Closest Pair problem



Convex Hull Problem



Important Problem Types: Numerical Problems

- Solving Equations
- Computing definite integrals
- Evaluating functions

Analysis Framework

What do you mean by analysing an algorithm?

Investigation of Algorithm's efficiency with respect to two resources

- Time
- Space

What is the need for Analysing an algorithm?

- To determine resource consumption
  - CPU time
  - Memory space
- Compare different methods for solving the same problem before actually implementing them and running the programs.
- To find an efficient algorithm

Complexity of an Algorithm

- A measure of the performance of an algorithm
- An algorithm's performance depends on
  - *internal* factors
    - Time required to run
    - Space (memory storage)required to run
  - *external* factors
    - Speed of the computer on which it is run
    - Quality of the compiler
    - Size of the input to the algorithm

Performance of Algorithm

important Criteria for performance:

- Space efficiency - the memory required, also called, space complexity
- Time efficiency - the time required, also called time complexity

Space Complexity

S(P)=C+SP(I)

- Fixed Space Requirements (C)
  Independent of the characteristics of the inputs and outputs
  - instruction space
  - space for simple variables, fixed-size structured variable, constants
- Variable Space Requirements (SP(I))
  dependent on the instance characteristic I
  - number, size, values of inputs and outputs associated with I
  - recursive stack space, formal parameters, local variables, return address

Space Complexity

$S(P)=C+S_P(I)$

```
float rsum(float list[ ], int n)
{
  if (n)
    return rsum(list, n-1) + list[n-1]
  return 0
 }
```

$S_{sum}(I)=S_{sum}(n)=6n$

Type	Name	Number of bytes
parameter: float parameter: integer return address:(used internally)	list [ ] n	2 2 2
TOTAL per recursive call		6

Time Complexity

$T(P)=C+T_P(I)$

- Compile time (C)
  independent of instance characteristics

- run (execution) time TP

PES UNIVERSITY

Experimental study

- Write a program implementing the algorithm
- Run the program with inputs of varying size and composition
- Get an accurate measure of the actual running time
- Use a method like System.currentTimeMillis()
- Plot the results

Limitations of Experimental study

- It is necessary to implement the algorithm, which may be difficult
- Results may not be indicative of the running time on other inputs not included in the experiment.
- In order to compare two algorithms, the same hardware and software environments must be used
- Experimental data though important is not sufficient

Theoretical Analysis

- Uses a high-level description of the algorithm instead of an implementation
- Characterizes running time as a function of the input size, n.
- Takes into account all possible inputs
- Allows us to evaluate the speed of an algorithm independent of the hardware/software environment

Theoretical Analysis

Two approaches:

1.Order of magnitude/asymptotic categorization –

This uses coarse categories and gives a general idea of performance.
If algorithms fall into the same category, if data size is small,
or if performance is critical, use method 2

2.Estimation of running time -

1. *operation counts* - select operation(s) that are executed most frequently and determine how many times each is done.
2. *step counts* - determine the total number of steps, possibly lines of code, executed by the program.

Analysis Framework

- Measuring an input's size
- Measuring running time
- Orders of growth (of the algorithm's efficiency function)
- Worst-base, best-case and average efficiency

Measuring an input's size

Efficiency is defined as a function of input size.

Input size depends on the problem.

Example 1, what is the input size of the problem of sorting n numbers?

Example 2, what is the input size of adding two n by n matrices?

Units for Measuring Running Time

- Measure the running time using standard unit of time measurements, such as seconds, minutes?
  Depends on the speed of the computer.

- count the number of times each of an algorithm's operations is executed.
  (step count method)
  Difficult and unnecessary

- count the number of times an algorithm's basic operation is executed.

  Basic operation: the most important operation of the algorithm, the operation contributing the most to the total running time.

  For example, the basic operation is usually the most time-consuming operation in the algorithm's innermost loop.

Analysis in the RAM Model

SmartFibonacci(n)		*cost*	*times* ($n > 1$)
1	**if** $n = 0$	c_1	1
2	**then return** 0	c_2	0
3	**elseif** $n = 1$	c_3	1
4	**then return** 1	c_4	0
5	**else** $pprev \leftarrow 0$	c_5	1
6	$prev \leftarrow 1$	c_6	1
7	**for** $i \leftarrow 2$ **to** n	c_7	n
8	**do** $f \leftarrow prev + pprev$	c_8	$n-1$
9	$pprev \leftarrow prev$	c_9	$n-1$
10	$prev \leftarrow f$	c_{10}	$n-1$
11	**return** f	c_{11}	1

$$T(n) = c_1 + c_3 + c_5 + c_6 + c_{11} + nc_7 + (n-1)(c_8 + c_9 + c_{10})$$

$$T(n) = nC_1 + C_2 \quad \Rightarrow T(n) \text{ is a } \textit{linear function} \text{ of } n$$

Measuring Running Time: Basic operation count

Input Size and Basic Operation Examples

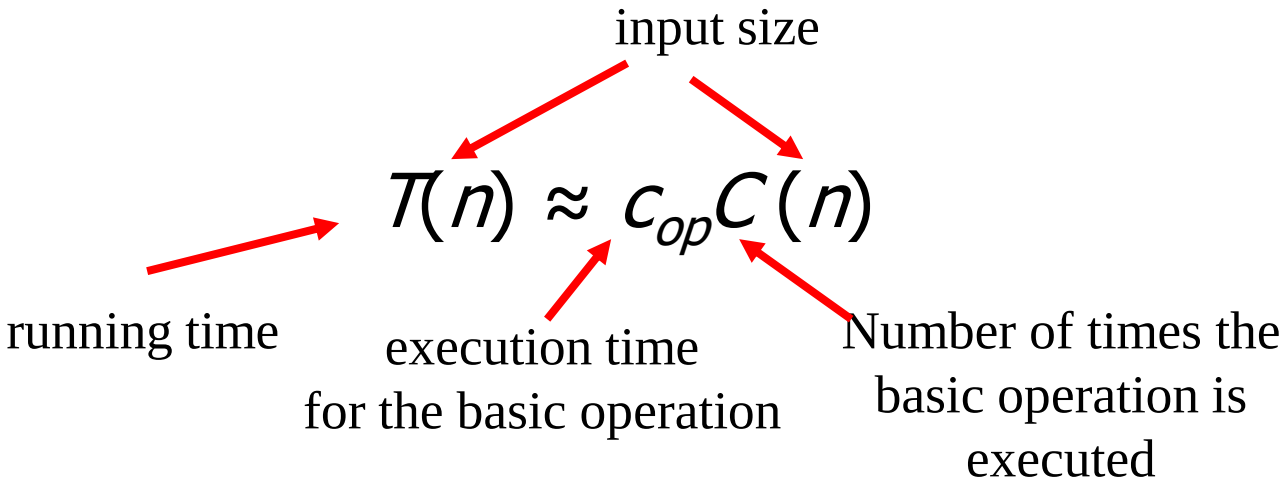
Problem	***Input size measure***	***Basic operation***
Search for a key in a list of *n* items	Number of items in list, *n*	Key comparison
Add two *n* by *n* matrices	Dimensions of matrices, *n*	addition
multiply two matrices	Dimensions of matrices, *n*	multiplication

Theoretical Analysis of Time Efficiency: Basic operation count

Time efficiency is analyzed by determining the number of repetitions of the basic operation as a function of input size.

Assuming C(n) = (1/2)n(n-1),

how much longer will the algorithm run if we double the input size?



Order of Growth

C(n) Basic Operation Count

- The efficiency analysis framework ignores the multiplicative constants of C(n) and focuses on the orders of growth of the C(n).

- Simple characterization of the algorithm's efficiency by identifying relatively significant term in the C(n).

Order of Growth

Why do we care about the order of growth of an algorithm's efficiency function, i.e., the total number of basic operations?

- Because, for smaller inputs, it is difficult to distinguish inefficient algorithms vs. efficient ones.
- For example, if the number of basic operations of two algorithms to solve a particular problem are n and n^2 respectively, then
  - if $n = 2$, Basic operation will be executed 2 and 4 times respectively for algorithm1 and 2.

    Not much difference!!!
  - On the other hand, if $n = 10000$, then it does makes a difference whether the number of times the basic operation is executed is n or n^2.

Order of Growth

n	$\log_2 n$	n	$n \log_2 n$	n^2	n^3	2^n	$n!$
10	3.3	10^1	$3.3 \cdot 10^1$	10^2	10^3	10^3	$3.6 \cdot 10^6$
10^2	6.6	10^2	$6.6 \cdot 10^2$	10^4	10^6	$1.3 \cdot 10^{30}$	$9.3 \cdot 10^{157}$
10^3	10	10^3	$1.0 \cdot 10^4$	10^6	10^9		
10^4	13	10^4	$1.3 \cdot 10^5$	10^8	10^{12}		
10^5	17	10^5	$1.7 \cdot 10^6$	10^{10}	10^{15}		
10^6	20	10^6	$2.0 \cdot 10^7$	10^{12}	10^{18}		

Exponential-growth functions

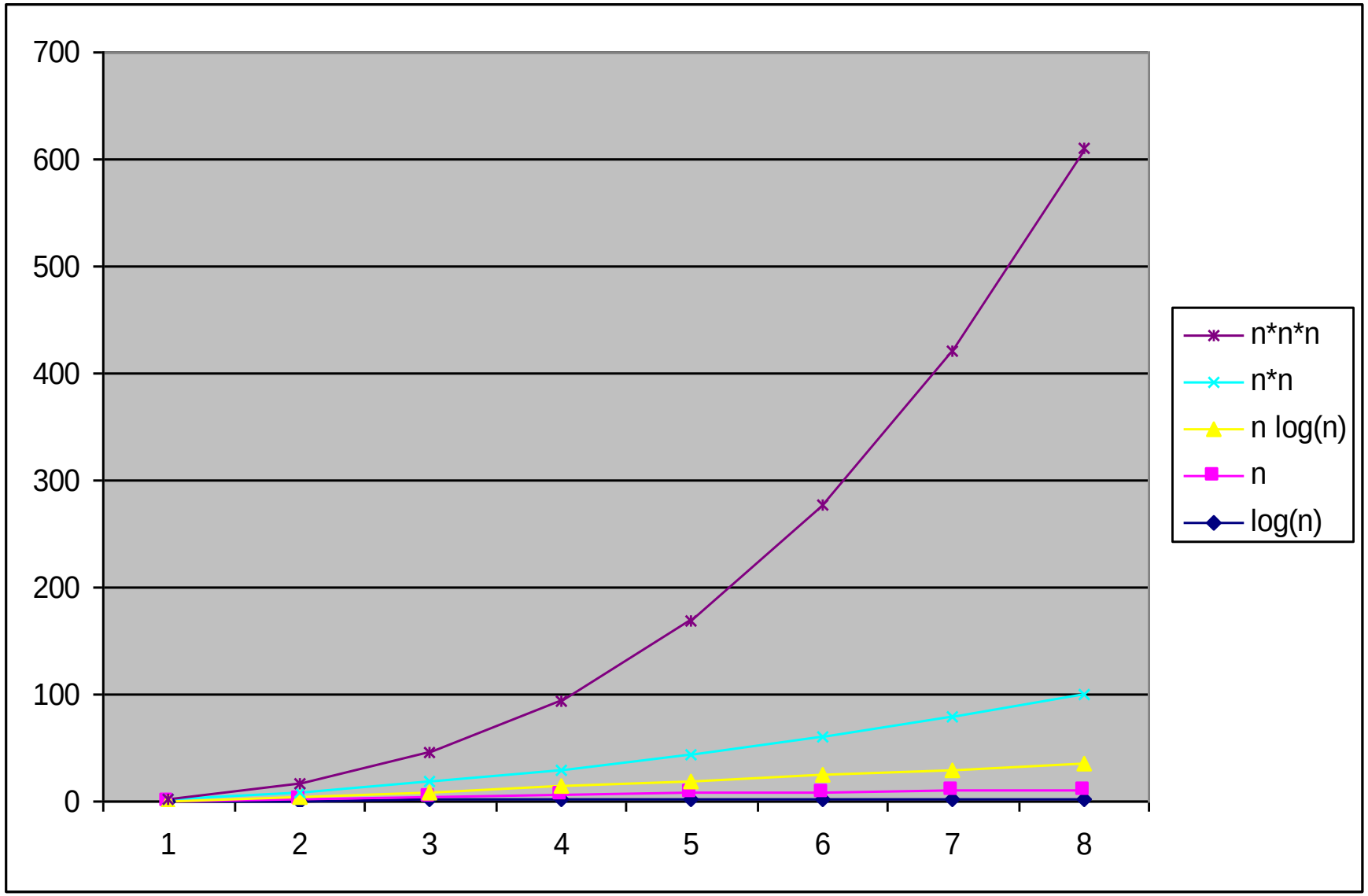
Table 2.1 Values (some approximate) of several functions important for analysis of algorithms

Orders of growth:

- consider only the leading term of a formula
- ignore the constant coefficient.

Design and Analysis of Algorithms

Order of Growth



Basic Efficiency Classes

1	constant
$\log n$	logarithmic
n	linear
$n \log n$	n-log-n
n^2	quadratic
n^3	cubic
2^n	exponential
$n!$	factorial

Best, Worst and Average case Analysis

- Algorithm efficiency depends on the input size n
- For some algorithms efficiency depends on type of input.

  Example: Sequential Search

  Problem: Given a list of n elements and a search key K, find an element equal to K, if any.

  Algorithm: Scan the list and compare its successive elements with K until either a matching element is found (successful search) or the list is exhausted (unsuccessful search)

Given a sequential search problem of an input size of n, what kind of input would make the running time the longest? How many key comparisons?

Best, Worst and Average case Analysis

- Worst case Efficiency
  - Efficiency (# of times the basic operation will be executed) for the worst case input of size n.
  - The algorithm runs the longest among all possible inputs of size n.
- Best case
  - Efficiency (# of times the basic operation will be executed) for the best case input of size n.
  - The algorithm runs the fastest among all possible inputs of size n.
- Average case:
  - Efficiency (#of times the basic operation will be executed) for a typical/random input of size n.
  - NOT the average of worst and best case
- How to find the average case efficiency?

Best, Worst and Average case Analysis

```
ALGORITHM SequentialSearch(A[0..n-1], K)
  //Searches for a given value in a given array by sequential
  search
  //Input: An array A[0..n-1] and a search key K
  //Output: Returns the index of the first element of A that
  matches K or –1 if there are no matching elements
  i ←0
  while i < n and A[i] ǂ K do
        i ← i + 1
  if i < n          //A[I] = K
        return i
  else
        return -1
```

Best, Worst and Average case Analysis: Sequential Search

- Worst-Case: Cworst(n) = n
- Best-Case: Cbest(n) = 1
- Average-Case

  from **(n+1)/2** to **(n+1)**

Average case Analysis: Sequential Search

Let **'p'** be the probability that key is found in the list

Assumption: All positions are equally probable

Case1: key is found in the list

$C_{avg,case1}(n) = p*(1 + 2 + \ldots + n) / n = p*(n + 1) / 2$

Case2: key is not found in the list

$C_{avg,\ case2}(n) = (1-p)*(n)$

$\mathbf{C_{avg}(n) = p}(n + 1) / 2 \mathbf{+ (1 - p)}(n)$

Orders of growth of an algorithm's basic operation count is important

How do we compare order of growth??

Using Asymptotic Notations

A way of comparing functions that ignores constant factors and small input sizes

- $O(g(n))$: class of functions $f(n)$ that grow *no faster* than $g(n)$
- $\Omega(g(n))$: class of functions $f(n)$ that grow *at least as fast* as $g(n)$
- $\Theta(g(n))$: class of functions $f(n)$ that grow *at same rate* as $g(n)$
- $o(g(n))$: class of functions $f(n)$ that grow *at slower rate* than $g(n)$
- $\omega(g(n))$: class of functions $f(n)$ that grow *at faster rate* than $g(n)$

O-notation

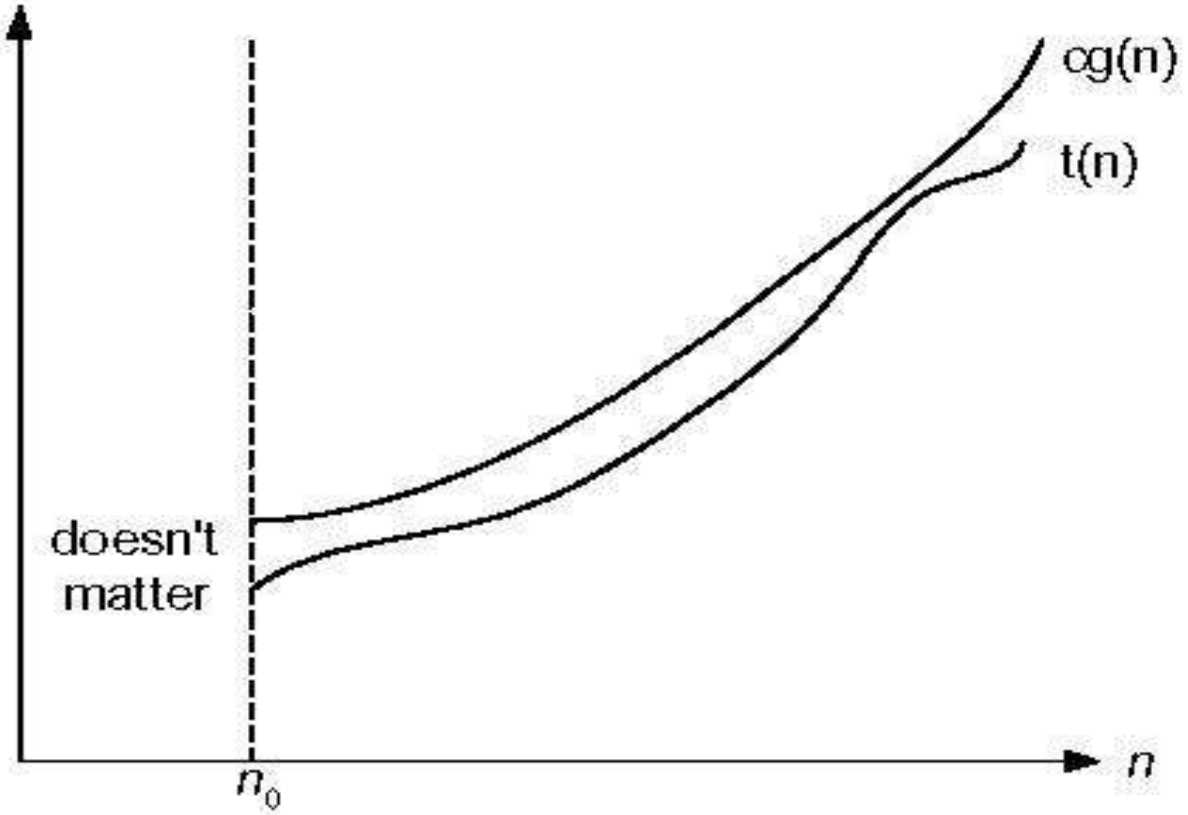


Figure 2.1 Big-oh notation: $t(n) \in O(g(n))$

O-notation

Formal definition

A function *t(n)* is said to be in *O(g(n)),* denoted *t(n)* $\in O(g(n))$,

if *t(n)* is bounded above by some constant multiple of *g(n)* for all large *n*,

i.e., if there exist some positive constant c and some nonnegative integer n_0 such that

$$t(n) \leq cg(n) \text{ for all } n \geq n_0$$

Exercises: prove the following using the above definition

$10n^2 \in O(n^2)$

$10n^2 + 2n \in O(n^2)$

$100n + 5 \in O(n^2)$

$5n+20 \in O(n)$

Ω-notation

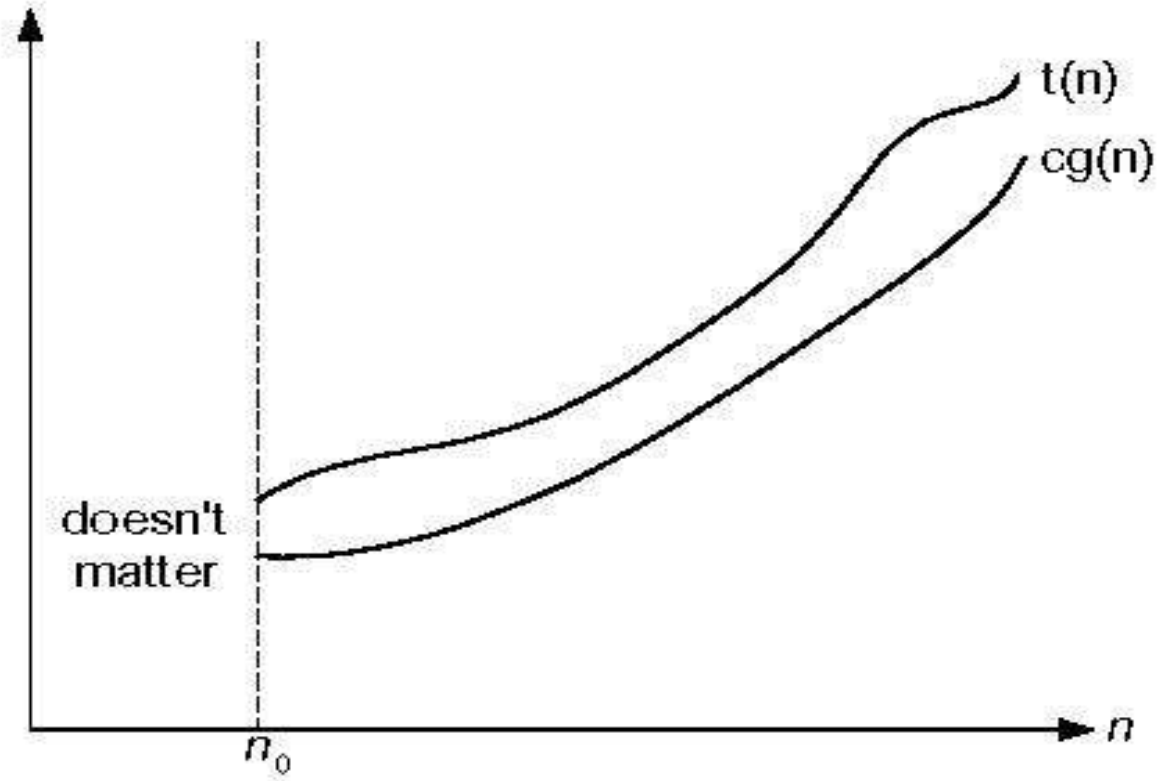


Fig. 2.2 Big-omega notation: $t(n) \in \Omega(g(n))$

Ω-notation

Formal definition

A function *t(n)* is said to be in $\Omega(g(n))$, denoted $t(n) \in \Omega(g(n))$, if *t(n)* is bounded below by some constant multiple of *g(n)* for all large *n*,

i.e., if there exist some positive constant c and some nonnegative integer n_0 such that

$t(n) \geq cg(n)$ for all $n \geq n_0$

Exercises: prove the following using the above definition

$10n^2 \in \Omega(n^2)$

$10n^2 + 2n \in \Omega(n^2)$

$10n^3 \in \Omega(n^2)$

Θ-notation

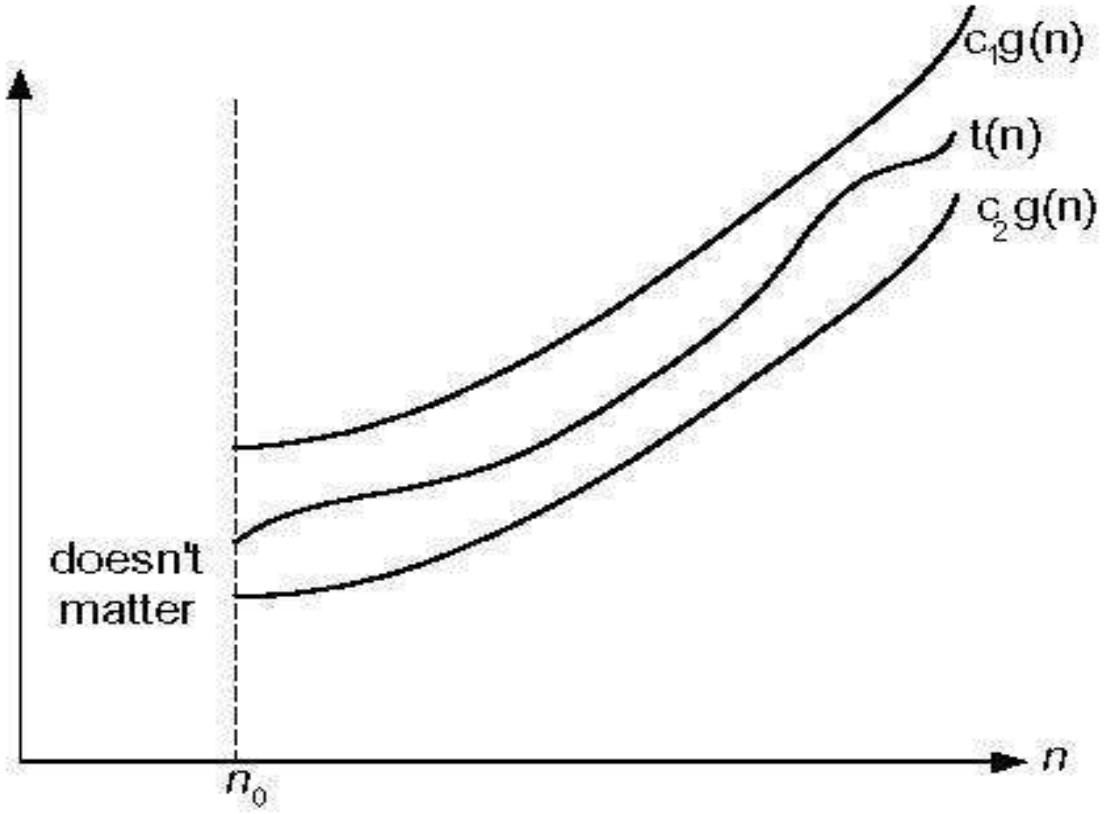


Figure 2.3 Big-theta notation: $t(n) \in \Theta(g(n))$

Θ-notation

Formal definition

A function *t(n)* is said to be in $\Theta(g(n))$, denoted $t(n) \in \Theta(g(n))$, if *t(n)* is bounded both above and below by some positive constant multiples of *g(n)* for all large *n*,

i.e., if there exist some positive constant c_1 and c_2 and some nonnegative integer n_0 such that

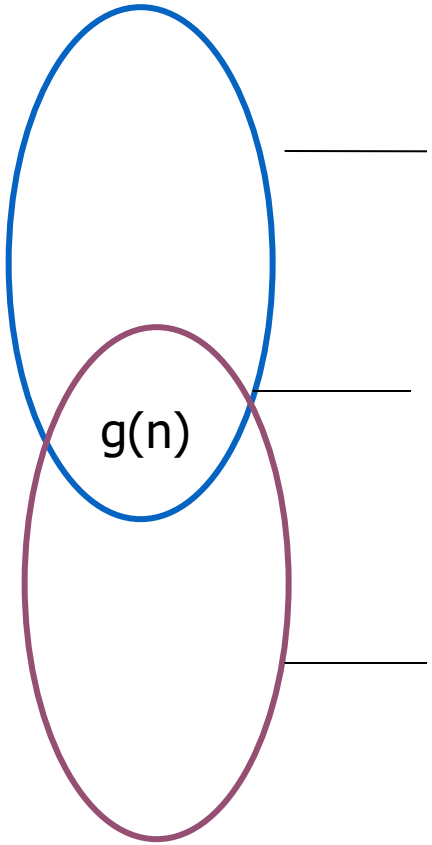
$$c_2\, g(n) \leq t(n) \leq c_1\, g(n) \text{ for all } n \geq n_0$$

Exercises: prove the following using the above definition

$10n^2 \in \Theta(n^2)$

$10n^2 + 2n \in \Theta(n^2)$

$(1/2)n(n-1) \in \Theta(n^2)$



>=

$\Omega(g(n))$, functions that grow at least as fast as g(n)

=

$\Theta(g(n))$, functions that grow at the same rate as g(n)

<=

O(g(n)), functions that grow no faster than g(n)

Little-o Notation

Formal Definition:

A function *t(n)* is said to be in Little-o*(g(n)),* denoted *t(n)* $\in o(g(n))$, if for any positive constant c and some nonnegative integer n_0

$0 \leq t(n) < cg(n)$ for all $n \geq n_0$

Little-o Notation

Example: $f(n) = 2n^2$ and $g(n) = n^2$ and $c = 2$

$f(n) = O(g(n))$ - Big-O

$f(n) \neq o(g(n))$ - little-o

If $f(n) = 2n$ & $g(n) = n^2$,

then for any value of $c>0$,

$\therefore$ $f(n) < c(n^2)$

$f(n) \neq o(g(n))$

Note : For non-negative functions, f(n) and g(n), f(n) is little o of g(n), if and only if,

$f(n) = O(g(n))$ and $f(n) \neq \theta(g(n))$

[strict upper bound, no lower bound]

Little Omega Notation

Formal Definition:

A function *t(n)* is said to be in Little- $\omega(g(n))$, denoted $t(n) \in \omega(g(n))$, if for any positive constant c and some nonnegative integer n_0

$t(n) > cg(n) \geq 0$ for all $n \geq n_0$

Little Omega Notation

Example : If $f(n) = 3n^2 + 2$, $g(n) = n$
then for any value of $c>0$
$f(n) > cg(n)$

$\therefore$ $f(n) = \omega(n)$

Note : For non-negative functions, f(n) *and* *g*(n),
f(n) is little ω of g(n), if and only if

$f(n) = \Omega(g(n))$ and $f(n) \neq \theta(g(n))$
[strict lower bound, no upper bound]

Theorems

- If $t_1(n) \in O(g_1(n))$ and $t_2(n) \in O(g_2(n))$, then
  $t_1(n) + t_2(n) \in O(\max\{g_1(n), g_2(n)\})$.
  For example,
  $5n^2 + 3n\log n \in O(n^2)$

- If $t1(n) \in \Theta(g1(n))$ and $t2(n) \in \Theta(g2(n))$, then
  $t1(n) + t2(n) \in \Theta(\max\{g1(n), g2(n)\})$

- $t1(n) \in \Omega(g1(n))$ and $t2(n) \in \Omega(g2(n))$, then
  $t1(n) + t2(n) \in \Omega(\max\{g1(n), g2(n)\})$

Implication: The algorithm's overall efficiency will be determined by the part with a larger order of growth, I.e., its least efficient part.

Design and Analysis of Algorithms

Using Limits to Compare Order of Growth

$$\lim_{n\to\infty} \frac{t(n)}{g(n)} = \begin{cases} 0 & \text{implies that } t(n) \text{ has a smaller order of growth than } g(n), \\ c & \text{implies that } t(n) \text{ has the same order of growth as } g(n), \\ \infty & \text{implies that } t(n) \text{ has a larger order of growth than } g(n). \end{cases}$$

Case1: $t(n) \in O(g(n))$

Case2: $t(n) \in \Theta(g(n))$

Case3: $g(n) \in O(t(n))$

$t'(n)$ and $g'(n)$ are first-order derivatives of $t(n)$ and $g(n)$

L'Hopital's Rule $\lim_{n\to\infty} \frac{t(n)}{g(n)} = \lim_{n\to\infty} \frac{t'(n)}{g'(n)}$

Stirling's Formula $n! \approx \sqrt{2\pi n}\left(\frac{n}{e}\right)^n$ for large values of n

Using Limits to Compare Order of Growth: Example 1

Compare the order of growth of f(n) and g(n) using method of limits

$t(n) = 5n3 + 6n + 2\,, \qquad g(n) = n4$

$$\lim_{n\to\infty} \frac{t(n)}{g(n)} = \lim_{n\to\infty} \frac{5n^3+6n+2}{n^4} = \lim_{n\to\infty}\left(\frac{5}{n}+\frac{6}{n^3}+\frac{2}{n^4}\right) = 0$$

As per case1

t(n) = O(g(n))

$5n^3 + 6n + 2 = O(n^4)$

Using Limits to Compare Order of Growth: Example 2

$$t(n) = \sqrt{5n^2 + 4n + 2}$$

using the Limits approach determine g(n) such that f(n) = Θ(g(n))

Leading term in square root n2

$$g(n) = \sqrt{n^2} = n$$

$$\lim_{n\to\infty} \frac{t(n)}{g(n)} = \lim_{n\to\infty} \frac{\sqrt{5n^2 + 4n + 2}}{\sqrt{n^2}}$$

$$= \lim_{n\to\infty} \sqrt{\frac{5n^2 + 4n + 2}{n^2}} = \lim_{n\to\infty} \sqrt{5 + \frac{4}{n} + \frac{2}{n^2}} = \sqrt{5}$$

non-zero constant

Hence, t(n) = Θ(g(n)) = Θ(n)

Design and Analysis of Algorithms

Using Limits to Compare Order of Growth

$$lim_{n \to \infty} t(n)/g(n) \neq 0, \infty \Rightarrow t(n) \in \Theta(g(n))$$

$$lim_{n \to \infty} t(n)/g(n) \neq \infty \Rightarrow t(n) \in O(g(n))$$

$$lim_{n \to \infty} t(n)/g(n) \neq 0 \Rightarrow t(n) \in \Omega(g(n))$$

$$lim_{n \to \infty} t(n)/g(n) = 0 \Rightarrow t(n) \in o(g(n))$$

$$lim_{n \to \infty} t(n)/g(n) = \infty \Rightarrow t(n) \in \omega(g(n))$$

Using Limits to Compare Order of Growth: Example 3

Compare the order of growth of t(n) and g(n) using method of limits

$t(n) = \log_2 n,\ g(n) = \sqrt{n}$

$$\lim_{n\to\infty} \frac{\log_2 n}{\sqrt{n}} = \lim_{n\to\infty} \frac{(\log_2 n)'}{(\sqrt{n})'} = \lim_{n\to\infty} \frac{(\log_2 e)\frac{1}{n}}{\frac{1}{2\sqrt{n}}} = 2\log_2 e \lim_{n\to\infty} \frac{1}{\sqrt{n}} = 0$$

$$\log_2 n \in o(\sqrt{n})$$

Design and Analysis of Algorithms
Orders of growth of some important functions

- All logarithmic functions $\log_a n$ belong to the same class

  $\Theta(\log n)$ no matter what the logarithm's base $a > 1$ is

  $\log_{10} n \in \Theta(\log_2 n)$

- All polynomials of the same degree k belong to the same class:

  $a_k n^k + a_{k-1} n^{k-1} + \ldots + a_0 \in \Theta(n^k)$

- Exponential functions an have different orders of growth for different a's

  $3^n \notin \Theta(2^n)$

- order log n < order n^α ($\alpha>0$) < order a^n < order n! < order n^n

How to Establish Orders of Growth of an Algorithm's Basic Operation Count

Summary

- Method 1: Using limits.
  - L' Hôpital's rule
- Method 2: Using the theorem.
- Method 3: Using the definitions of O-, Ω-, and Θ-notation.

Time Efficiency of Non-recursive Algorithms

Steps in mathematical analysis of non-recursive algorithms:

- Decide on parameter n indicating input size
- Identify algorithm's basic operation
- Check whether the number of times the basic operation is executed depends only on the input size n. If it also depends on the type of input, investigate worst, average, and best case efficiency separately.
- Set up summation for C(n) reflecting the number of times the algorithm's basic operation is executed.
- Simplify summation using standard formulas

Useful Summation Formulas and Rules

$\Sigma_{l \le i \le u} 1 = 1+1+\ldots+1 = u - l + 1$

$\Sigma_{1 \le i \le n} i = 1+2+\ldots+n = n(n+1)/2$

$\Sigma_{1 \le i \le n} i^2 = 1^2+2^2+\ldots+n^2 = n(n+1)(2n+1)/6$

$\Sigma_{0 \le i \le n} a^i = 1 + a + \ldots + a^n = (a^{n+1} - 1)/(a - 1)$ for any $a \neq 1$

$\Sigma(a_i \pm b_i) = \Sigma a_i \pm \Sigma b_i \qquad \Sigma c a_i = c \Sigma a_i$

$\Sigma_{l \le i \le u} a_i = \Sigma_{l \le i \le m} a_i + \Sigma_{m+1 \le i \le u} a_i$

$$\sum_{i=l}^{u} 1 = (u - l + 1)$$

Example 1: Finding Max Element in a list

Algorithm *MaxElement (A[0..n-1])*
//Determines the value of the largest element
in a given array
//Input: An array A[0..n-1] of real numbers
//Output: The value of the largest element in A
maxval ← A[0]
for i ← 1 to n-1 do
    if A[i] > maxval
        maxval ← A[i]
return maxval

- The basic operation- comparison
- Number of comparisons is the same for all arrays of size n.
- Number of comparisons

$$C(n) = \sum_{i=1}^{n-1} 1 = n - 1 \in \Theta(n)$$

Example 2: Element Uniqueness Problem

Algorithm *UniqueElements (A[0..n-1])*
//Checks whether all the elements in a given
array are distinct
//Input: An array A[0..n-1]
//Output: Returns true if all the elements in A
are distinct and false otherwise
for i ← 0 to n - 2 do
    for j ← i + 1 to n – 1 do
        if A[i] = A[j] return false
return true

Best-case:

If the two first elements of the array are the same
No of comparisons in Best case = 1 comparison

Worst-case:

- Arrays with no equal elements
- Arrays in which only the last two elements are the pair of equal elements

Example 2: Element Uniqueness Problem

$$C_{worst}(n) = \sum_{i=0}^{n-2} \sum_{j=i+1}^{n-1} 1 = \sum_{i=0}^{n-2} [(n-1) - (i+1) + 1] = \sum_{i=0}^{n-2} (n-1-i)$$

$$= \sum_{i=0}^{n-2} (n-1) - \sum_{i=0}^{n-2} i = (n-1) \sum_{i=0}^{n-2} 1 - \frac{(n-2)(n-1)}{2}$$

$$= (n-1)^2 - \frac{(n-2)(n-1)}{2} = \frac{(n-1)n}{2} \approx \frac{1}{2} n^2$$

Best-case: 1 comparison

Worst-case: $n^2/2$ comparisons

$T(n)_{\text{worst case}} = O(n^2)$

Example 3:Matrix Multiplication

Algorithm *MatrixMultiplication(A[0..n-1, 0..n-1], B[0..n-1, 0..n-1])*

```
//Multiplies two square matrices of order n by the definition-based algorithm
//Input: two n-by-n matrices A and B
//Output: Matrix C = AB
for i ← 0 to n - 1 do
   for j ← 0 to n – 1 do
          C[i, j] ← 0.0
           for k ← 0 to n – 1 do
                    C[i, j] ← C[i, j] + A[i, k] * B[k, j]
return C
```

$M(n) \in \Theta(n^3)$

Steps in Mathematical Analysis of Recursive Algorithms

- Decide on parameter n indicating input size
- Identify algorithm's basic operation
- If the number of times the basic operation is executed varies with different inputs of same sizes , investigate worst, average, and best case efficiency separately
- Set up a recurrence relation and initial condition(s) for C(n)-the number of times the basic operation will be executed for an input of size n
- Solve the recurrence or estimate the order of magnitude of the solution

Important Recurrence Types

- Decrease-by-one recurrences

  A decrease-by-one algorithm solves a problem by exploiting a relationship between a given instance of size n and a smaller size $n - 1$.

  Example: n!

  The recurrence equation has the form

  $T(n) = T(n-1) + f(n)$

- Decrease-by-a-constant-factor recurrences

  A decrease-by-a-constant algorithm solves a problem by dividing its given instance of size n into several smaller instances of size n/b, solving each of them recursively, and then, if necessary, combining the solutions to the smaller instances into a solution to the given instance.

  Example: binary search.

  The recurrence has the form

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

- One (constant) operation reduces problem size by one.

  $T(n) = T(n-1) + c \qquad T(1) = d$

  Solution: $T(n) = (n-1)c + d$ ***linear***

- A pass through input reduces problem size by one.

  $T(n) = T(n-1) + c\,n \qquad T(1) = d$

  Solution: $T(n) = [n(n+1)/2 - 1]\,c + d$ ***quadratic***

Methods to solve recurrences

- Substitution Method
  - Mathematical Induction
  - Backward substitution
- Recursion Tree Method
- Master Method (Decrease by constant factor recurrences)

Design and Analysis of Algorithms

Recursive Evaluation of n!

$n! = 1 * 2 * \ldots * (n-1) * n$ for $n \geq 1$ and $0! = 1$

Recursive definition of $n!$:

$F(n) = F(n-1) * n$ for $n \geq 1$

$F(0) = 1$

input size?

basic operation?

Best/Worst/Average Case?

ALGORITHM $F(n)$

//Computes $n!$ recursively
//Input: A nonnegative integer n
//Output: The value of $n!$
if $n = 0$ **return** 1
else return $F(n-1) * n$

$$M(n) = M(n-1) + 1 \quad \text{for } n > 0,$$
$$M(0) = 0.$$

M(n-1) = M(n-2) + 1; **M(n-2) = M(n-3)+1**

M(n) = n

Overall time Complexity: Θ(n)

Design and Analysis of Algorithms

Counting number of binary digits in binary representation of a number

ALGORITHM *BinRec(n)*

```
//Input: A positive decimal integer n
//Output: The number of binary digits in n's binary representation
if n = 1 return 1
else return BinRec(⌊n/2⌋) + 1
```

input size?

basic operation?

Best/Worst/Average Case?

$$A(2^k) = A(2^{k-1}) + 1 \quad \text{for } k > 0,$$
$$A(2^0) = 0.$$

$$\begin{aligned}
A(2^k) &= A(2^{k-1}) + 1 && \text{substitute } A(2^{k-1}) = A(2^{k-2}) + 1\\
&= [A(2^{k-2}) + 1] + 1 = A(2^{k-2}) + 2 && \text{substitute } A(2^{k-2}) = A(2^{k-3}) + 1\\
&= [A(2^{k-3}) + 1] + 2 = A(2^{k-3}) + 3 && \dots\\
&\dots\\
&= A(2^{k-i}) + i\\
&\dots\\
&= A(2^{k-k}) + k.
\end{aligned}$$

$$A(n) = \log_2 n \in \Theta(\log n).$$

Tower of Hanoi

```
Algorithm TowerOfHanoi(n, Src, Aux, Dst)
    if (n = 0)
        return
    TowerOfHanoi(n-1, Src, Dst, Aux)
    Move disk n from Src to Dst
    TowerOfHanoi(n-1, Aux, Src, Dst)
```

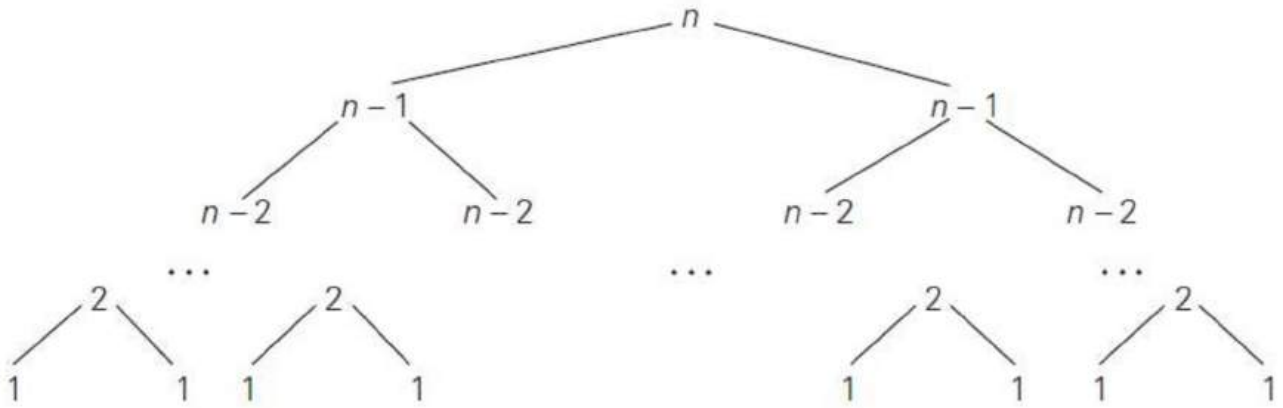
Input Size: `n`

Basic Operation : `Move disk n from Src to Dst`

C(n) = $\mathbf{2C(n-1) + 1}$ for n > 0 and C(0)=0

= $\mathbf{2^n - 1 \in \Theta(2^n)}$

Tower of Hanoi : Tree of Recursive calls



$$C(n) = \sum_{l=0}^{n-1} 2^l = 2^n - 1$$

Solving Recurrences: Example1

$T(n) = T(n-1) + 1 \quad n>0 \quad T(0) = 1$

$T(n) = T(n-1) + 1$

$= T(n-2) + 1 + 1 = T(n-2) + 2$

$= T(n-3) + 1 + 2 = T(n-3) + 3$

...

$= T(n-i) + i$

...

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

Solving Recurrences: Example2

$T(n) = T(n-1) + 2n - 1 \qquad T(0)=0$

$= [T(n-2) + 2(n-1) - 1] + 2n - 1$

$= T(n-2) + 2(n-1) + 2n - 2$

$= [T(n-3) + 2(n-2) - 1] + 2(n-1) + 2n - 2$

$= T(n-3) + 2(n-2) + 2(n-1) + 2n - 3$

...

$= T(n-i) + 2(n-i+1) + \ldots + 2n - i$

...

$= T(n-n) + 2(n-n+1) + \ldots + 2n - n$

$= 0 + 2 + 4 + \ldots + 2n - n$

$= 2 + 4 + \ldots + 2n - n$

$= 2*n*(n+1)/2 - n$

// arithmetic progression formula $1+\ldots+n = n(n+1)/2$ //

$= O(n^2)$

Solving Recurrences: Example3

$$T(n) = T(n/2) + 1 \quad n > 1$$
$$T(1) = 1$$

$$T(n) = T(n/2) + 1$$
$$= T(n/2^2) + 1 + 1$$
$$= T(n/2^3) + 1 + 1 + 1$$
.....
$$= T(n/2^i) + i$$
.....
$$= T(n/2^k) + k \quad (k = \log n)$$
$$= 1 + \log n$$
$$= O(\log\ n)$$

Design and Analysis of Algorithms

Solving Recurrences: Example4

$$T(n) = 2T(n/2) + cn \qquad n > 1 \qquad T(1) = c$$

$$T(n) = 2T(n/2) + cn$$
$$= 2(2T(n/2^2) + c(n/2)) + cn = 2^2 T(n/2^2) + cn + cn$$
$$= 2^2(2T(n/2^3) + c(n/2^2)) + cn + cn = 2^3 T(n/2^3) + 3cn$$
.....
$$= 2^i T(n/2^i) + icn$$
.....
$$= 2^k T(n/2^k) + kcn \quad (k = \log\ n)$$
$$= nT\ (1) + cn\log n = cn + cn\log n$$
$$= \mathrm{O}(n\log\ n)$$

Performance Evaluation of Algorithm

- Performance Analysis
  - Machine Independent
  - Prior Evaluation
- Performance Measurement
  - Machine Dependent
  - Posterior Evaluation

Design and Analysis of Algorithms

Performance Analysis of Sequential search :Worst Case

```
ALGORITHM SequentialSearch(A[0..n-1], K)
  //Searches for a given value in a given array by sequential search
  //Input: An array A[0..n-1] and a search key K
  //Output: Returns the index of the first element of A that matches K or –1 if there are no matching elements
  i ←0
  while i < n and A[i] ‡ K do
        i ← i + 1
  if i < n          //A[i] = K
        return i
  else
        return -1
```

Basic operation: A[i] ‡ K

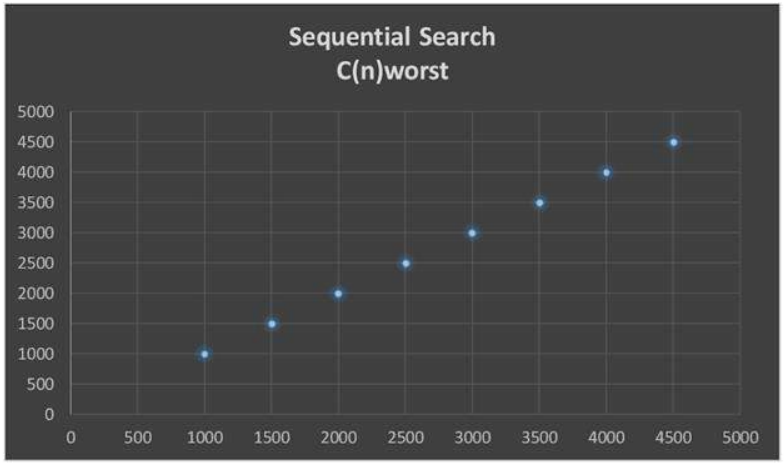
Basic operation count: n

Time Complexity: $T(n) \in O(n)$

Design and Analysis of Algorithms

Performance Analysis of Sequential Search

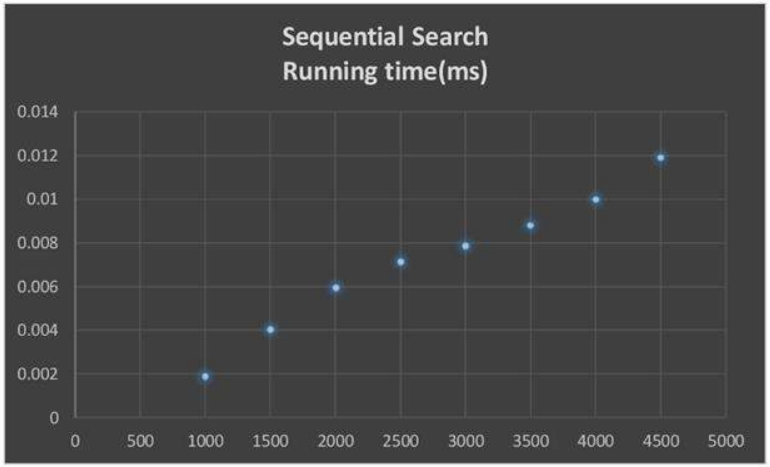
Input Size	Sequential Search C(n)worst
1000	1000
1500	1500
2000	2000
2500	2500
3000	3000
3500	3500
4000	4000
4500	4500



Design and Analysis of Algorithms

Performance Measurement of Sequential Search

Input Size	Sequential Search Actual Running Time(ms)
1000	0.001907
1500	0.004053
2000	0.00596
2500	0.007153
3000	0.007868
3500	0.008821
4000	0.010014
4500	0.011921



Brute Force

- In computer science, **brute-force search** or **exhaustive search**, also known as **generate and test**, is a very general problem-solving technique and algorithmic paradigm that consists of:
  - systematically enumerating all possible candidates for the solution
  - checking whether each candidate satisfies the problem's statement

https://en.wikipedia.org/wiki/Brute-force_search

Brute Force

- A brute-force algorithm to find the divisors of a natural number n would
  - enumerate all integers from 1 to n
  - check whether each of them divides n without remainder
- A brute-force approach for the eight queens puzzle would
  - examine all possible arrangements of 8 pieces on the 64-square chessboard
  - check whether each (queen) piece can attack any other, for each arrangement
- The brute-force method for finding an item in a table (linear search)
  - checks all entries of the table, sequentially, with the item

Brute Force

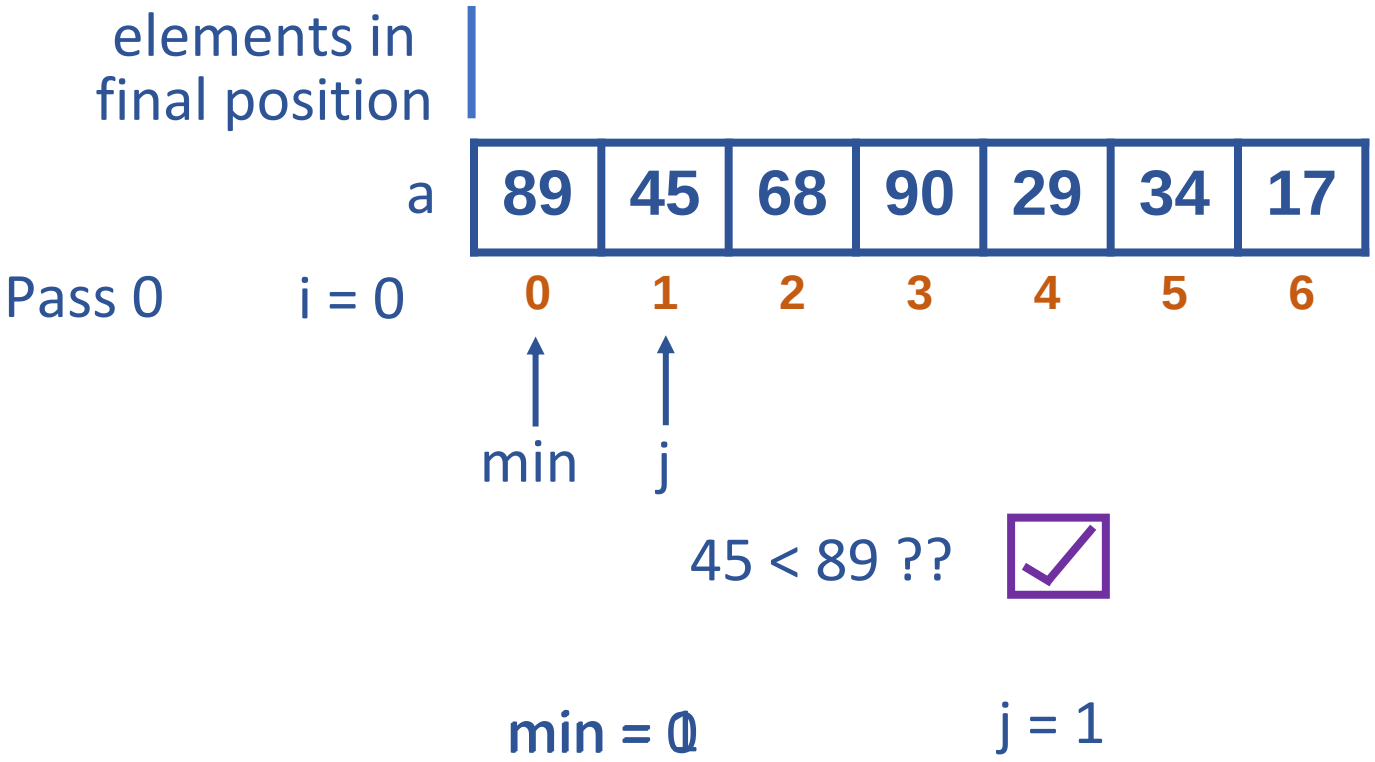
- A brute-force search is simple to implement, and will always find a solution if it exists
- But, its cost is proportional to the number of candidate solutions – which in many practical problems tends to grow very quickly as the size of the problem increases (Combinatorial explosion)
- Brute-force search is typically used
  - when the problem size is limited
  - when there are problem-specific heuristics that can be used to reduce the set of candidate solutions to a manageable size
  - when the simplicity of implementation is more important than speed

Selection Sort

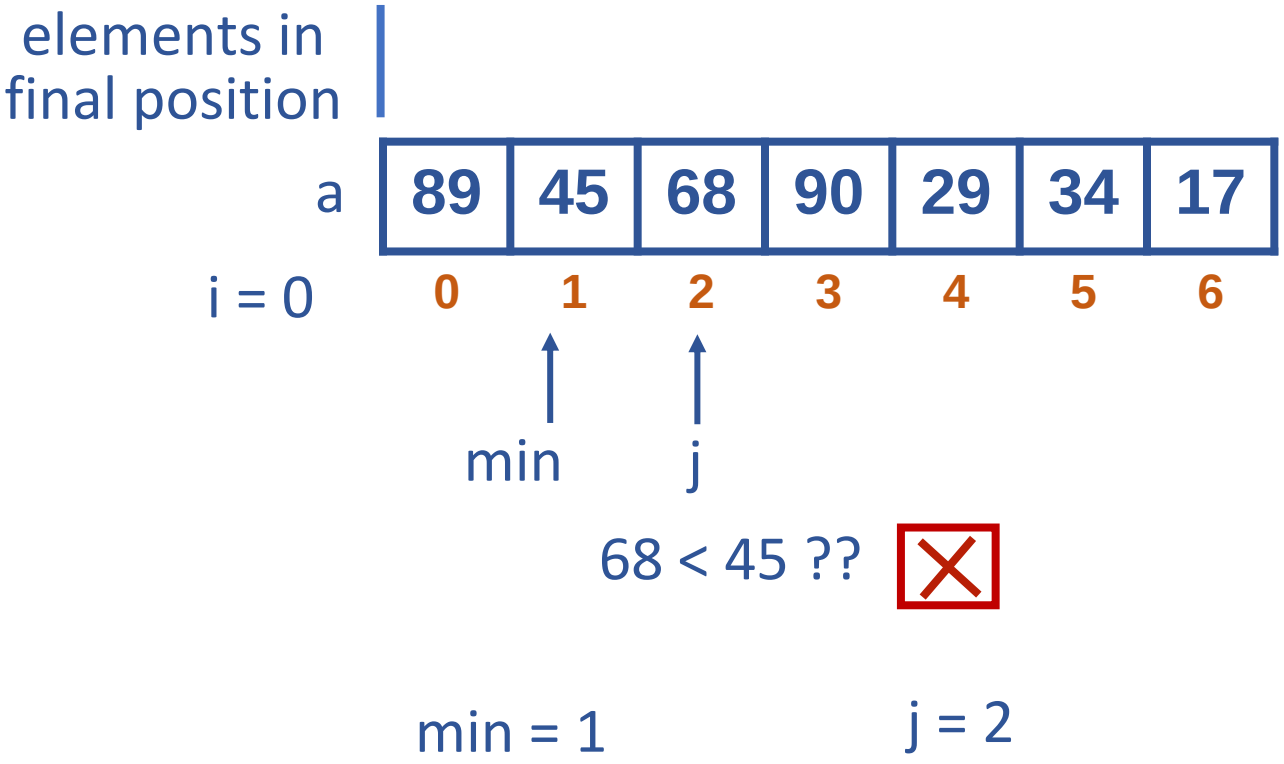
- Scan the array to find its smallest element and swap it with the first element, putting the smallest element in its final position in the sorted list
- Then, starting with the second element, scan the elements to the right of it to find the smallest among them and swap it with the second element, putting the second smallest element in its final position
- Generally, on pass i ($0 \le i \le n-2$), find the smallest element in $A[i..n-1]$ and swap it with $A[i]$

- $A[0] \le A[1] \le A[2] \ldots \le A[i-1]$ | A[i], ….., A[min], …., A[n-1]
- in their final positions the last n – i elements

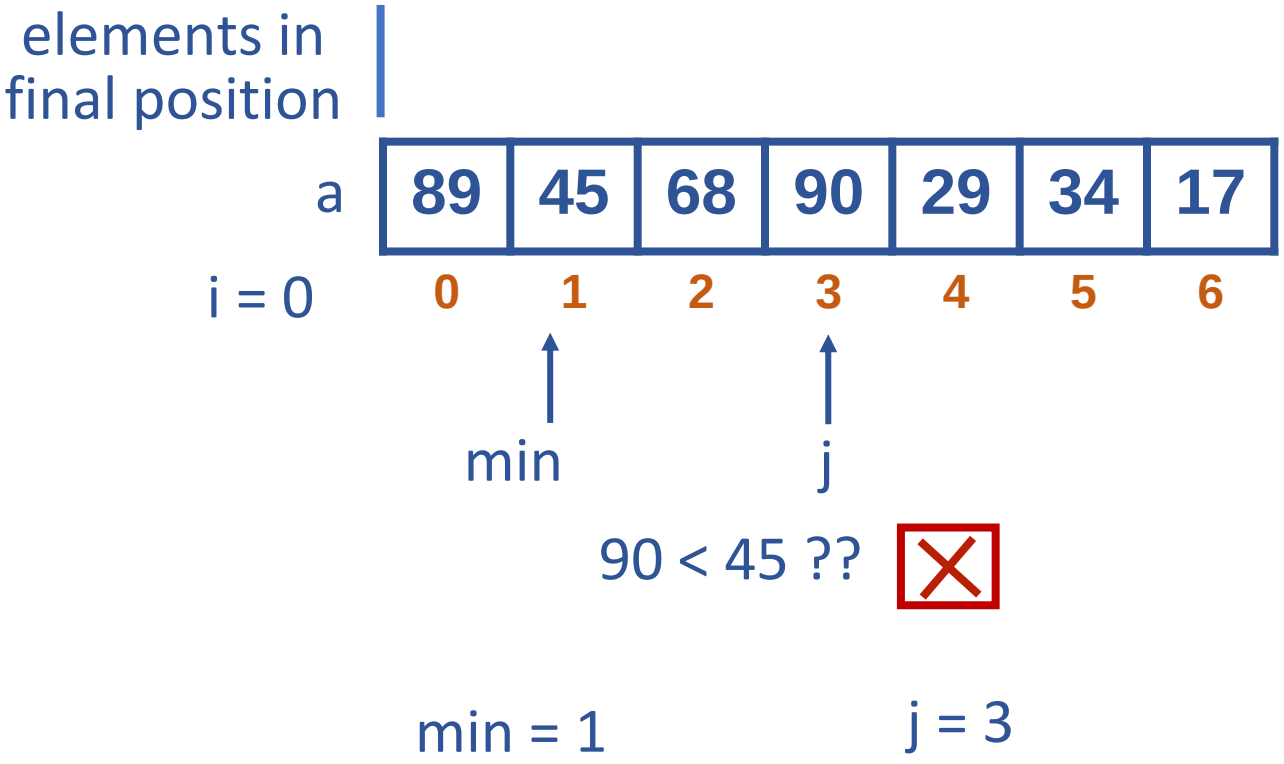
Selection Sort



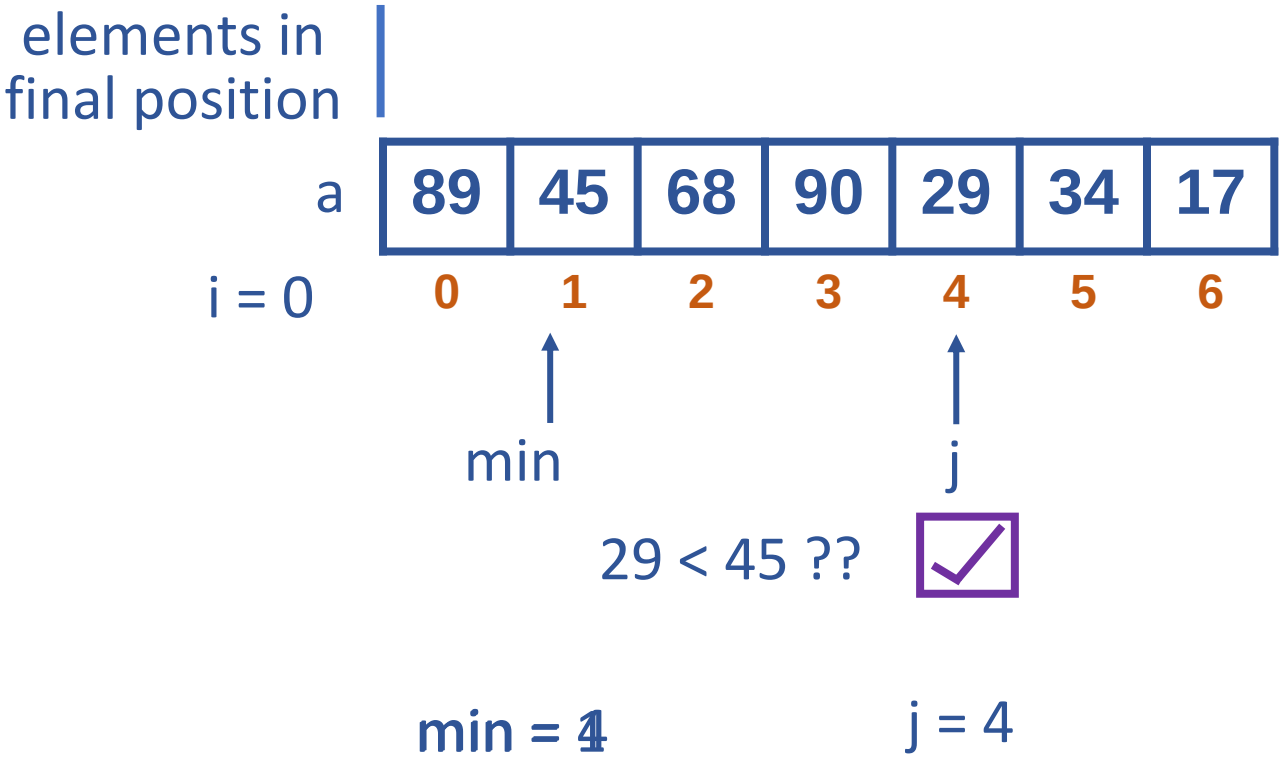
Selection Sort



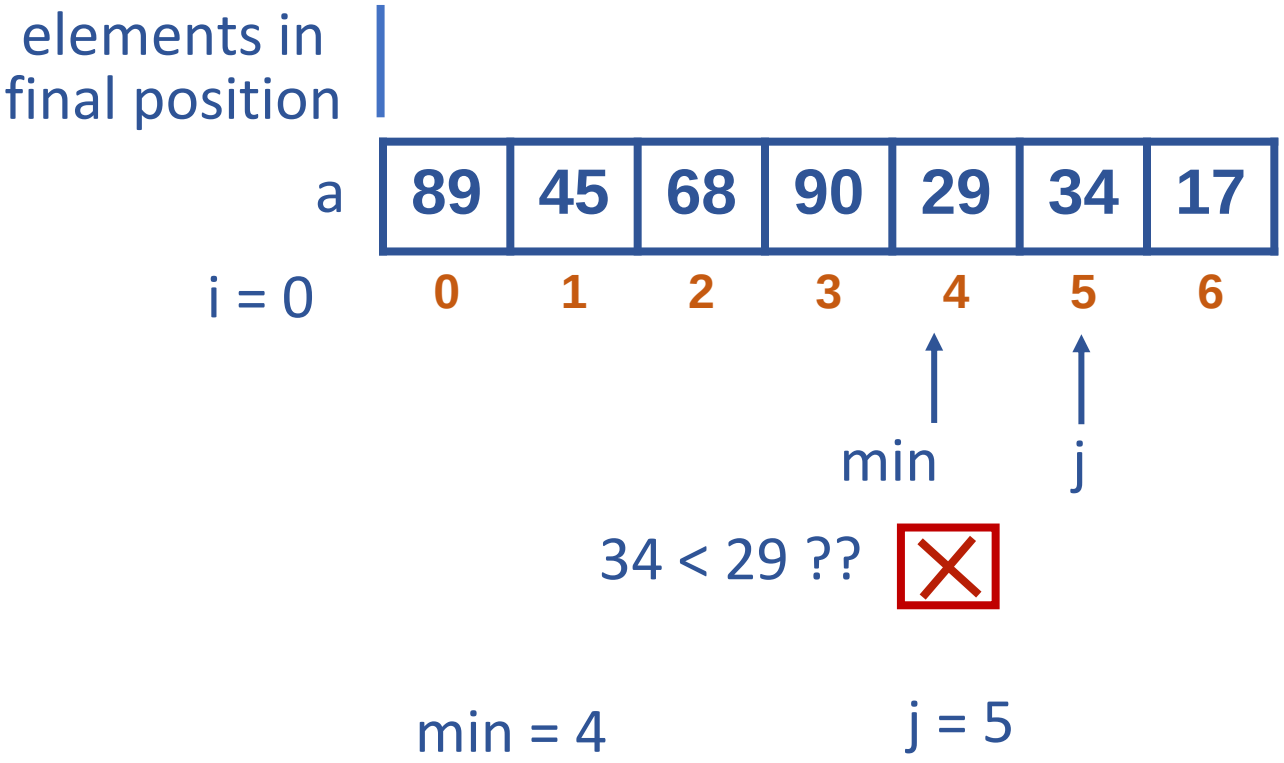
Selection Sort



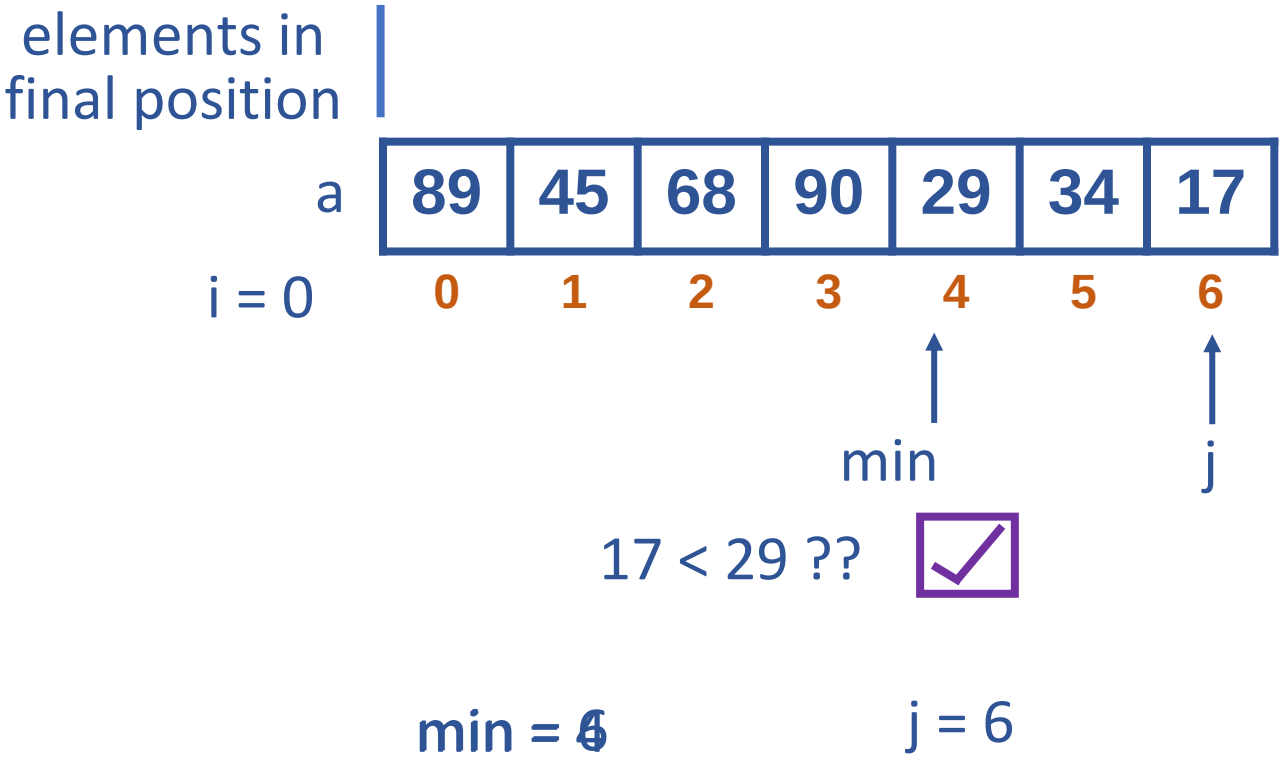
Selection Sort



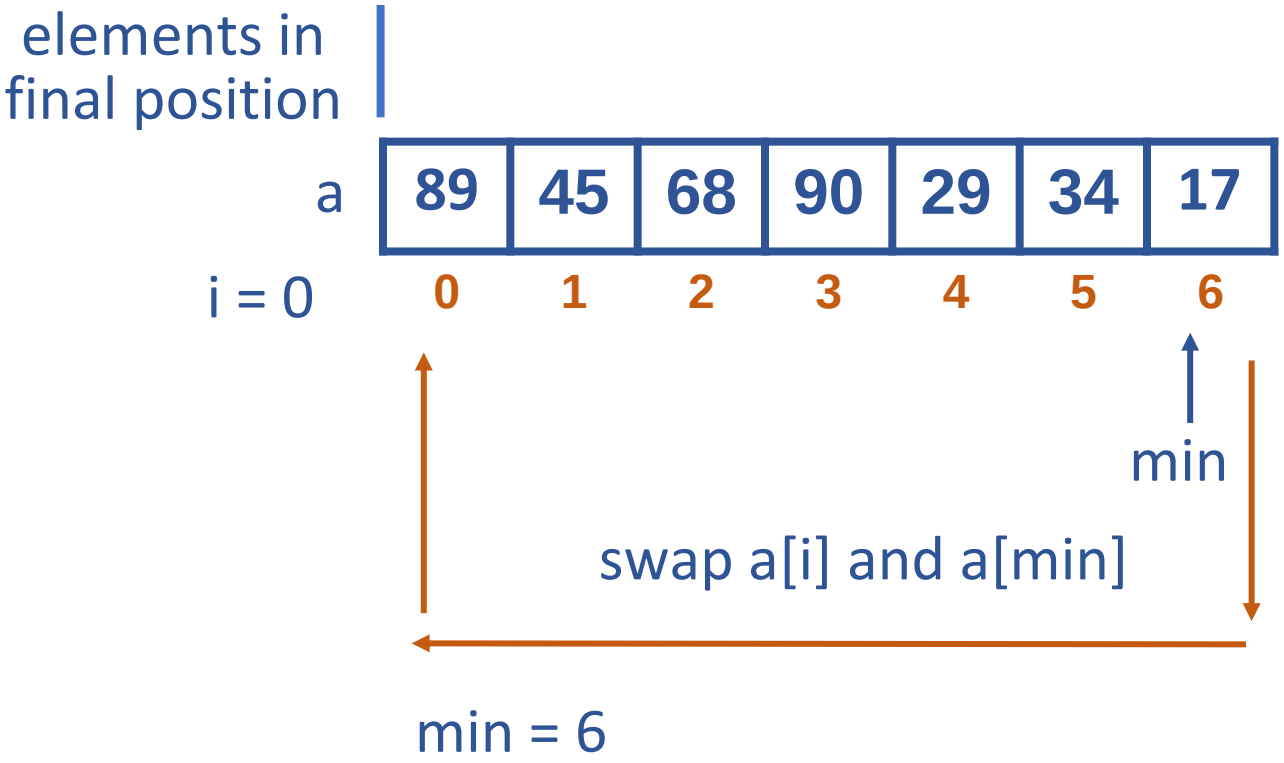
Selection Sort



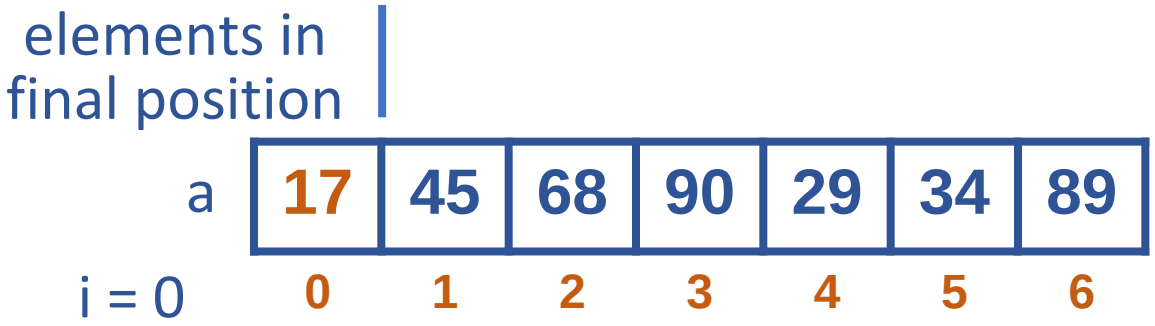
Selection Sort



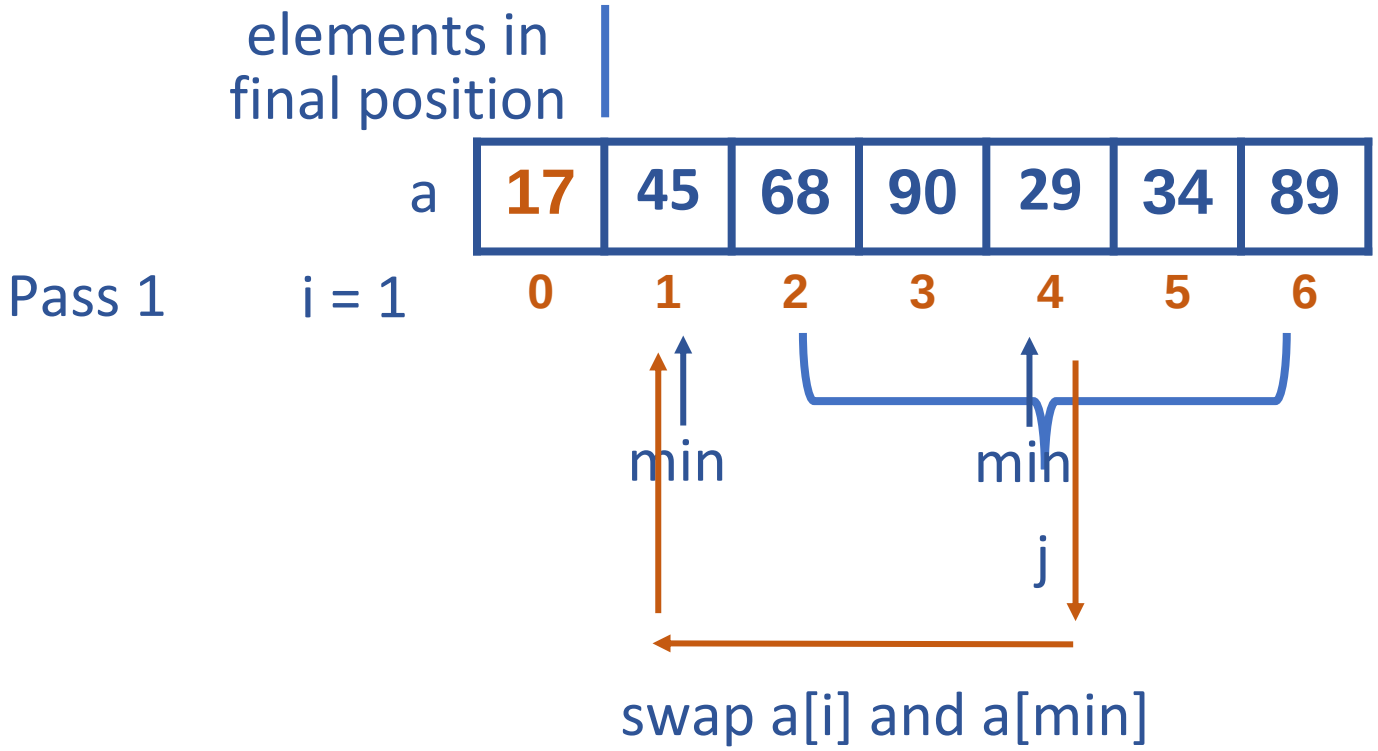
Selection Sort



Selection Sort

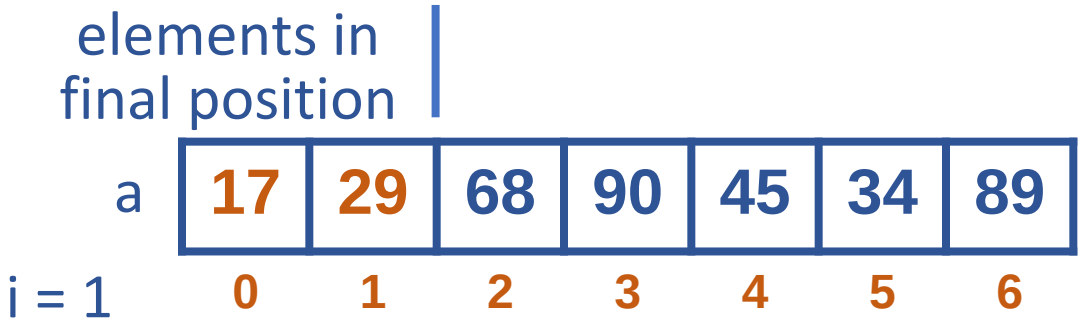


Selection Sort

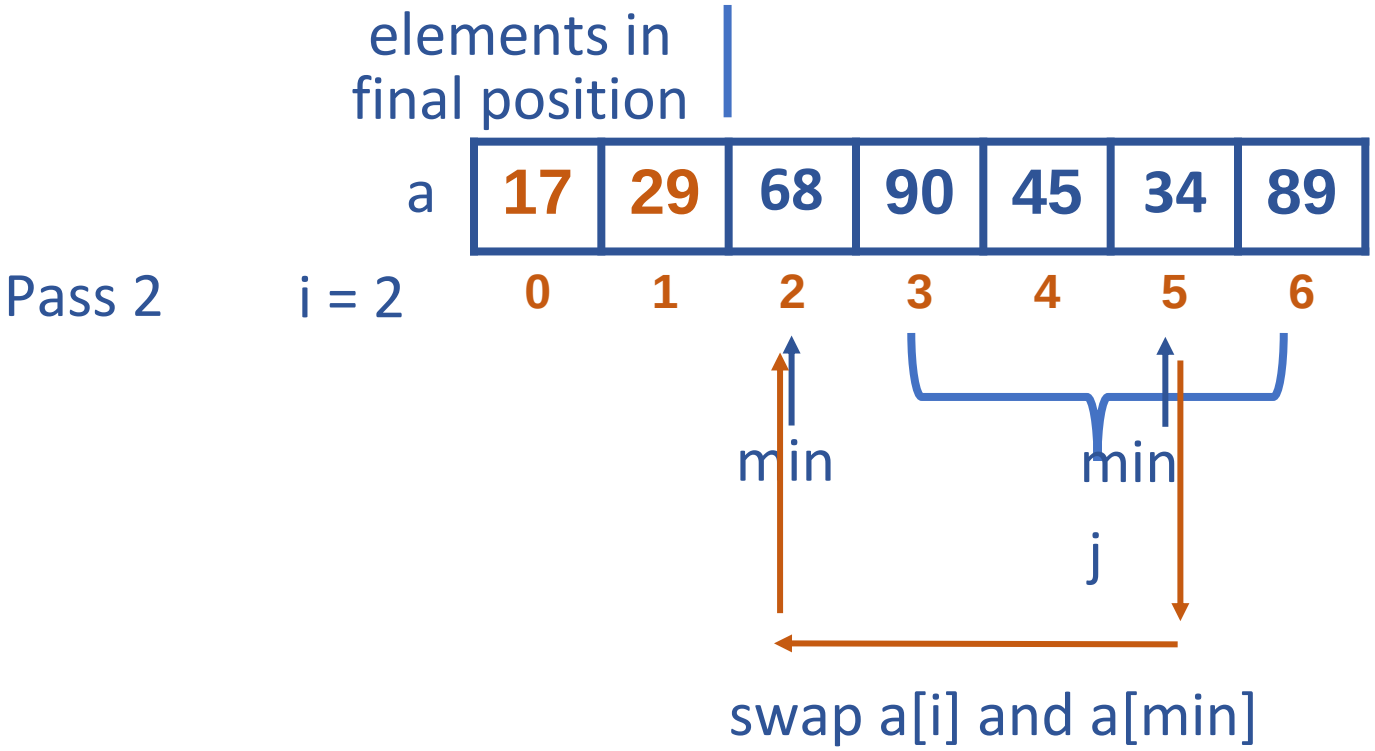


min = 4

Selection Sort

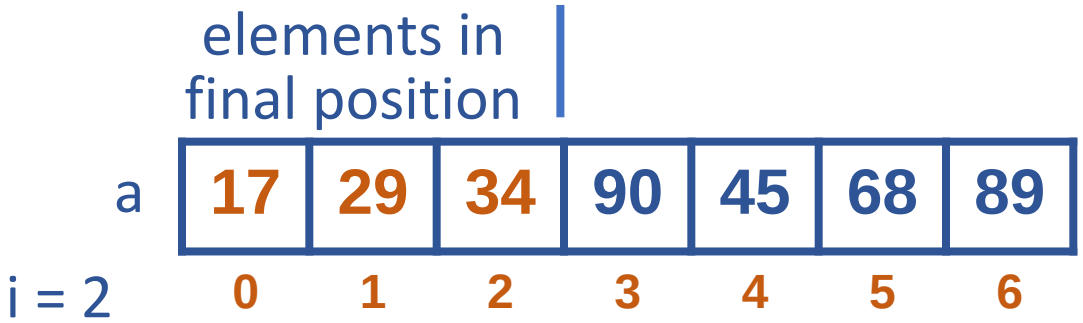


Selection Sort

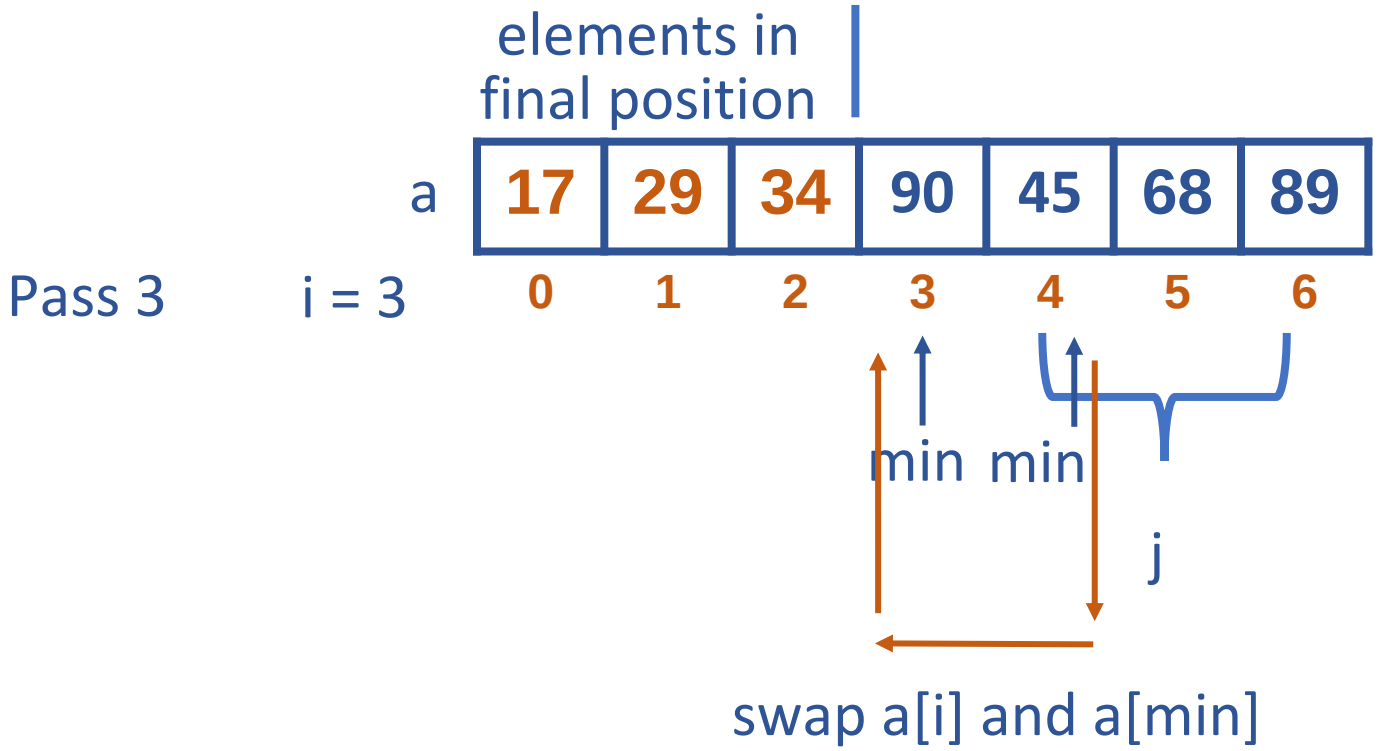


min = 5

Selection Sort

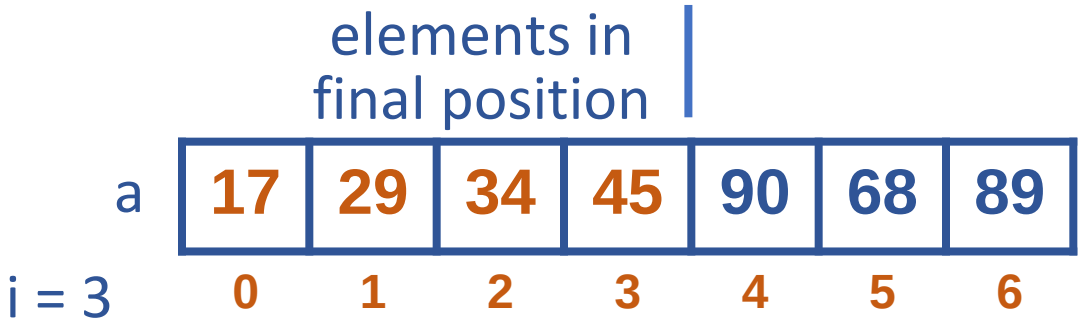


Selection Sort

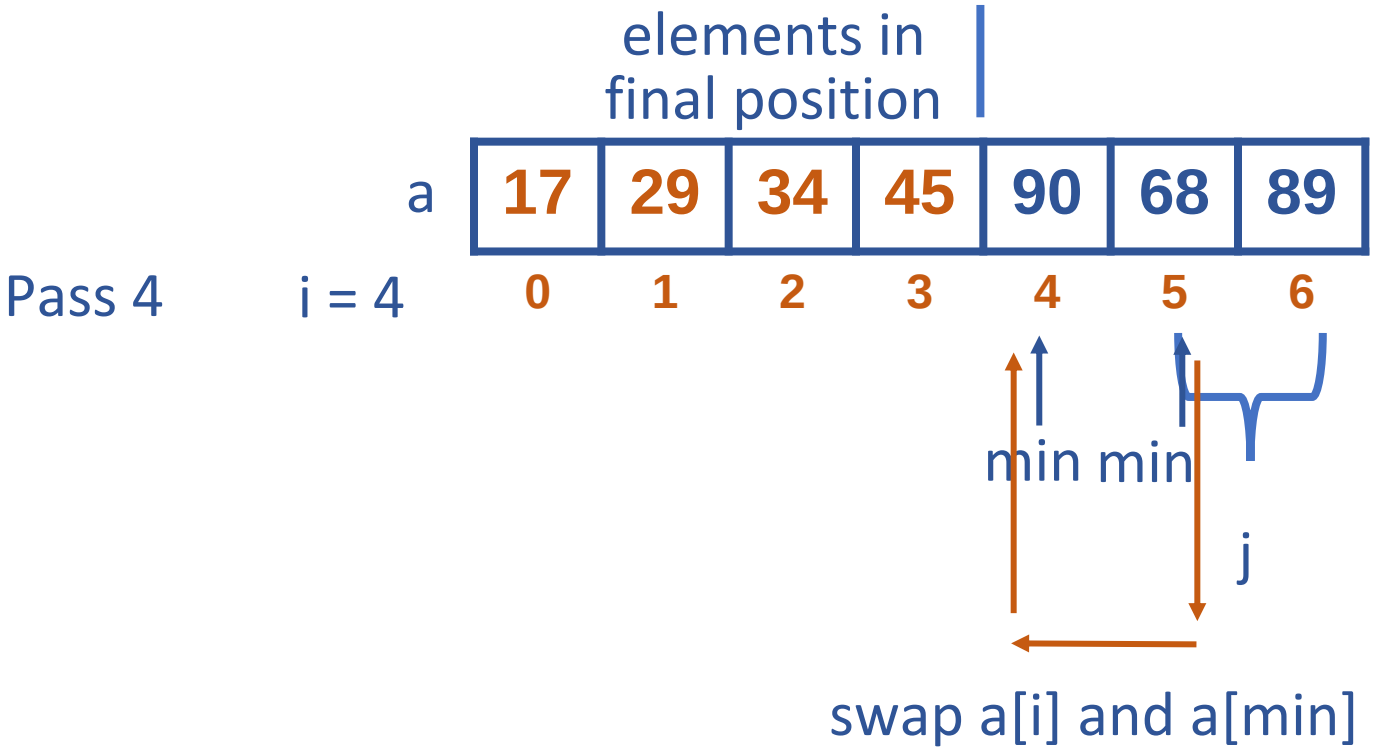


min = 4

Selection Sort

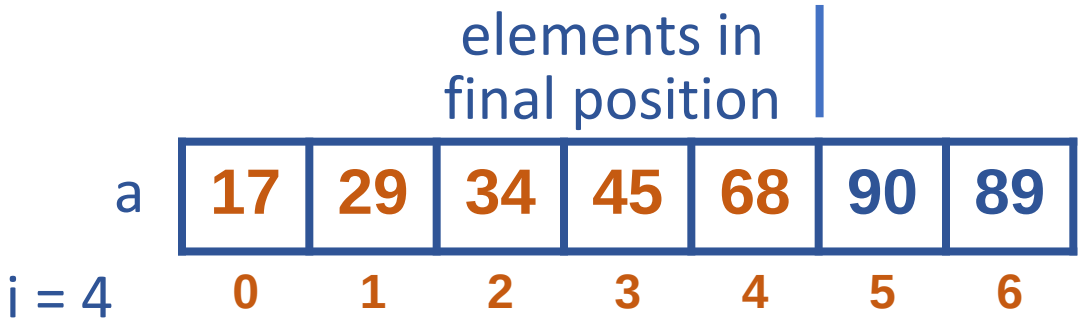


Selection Sort

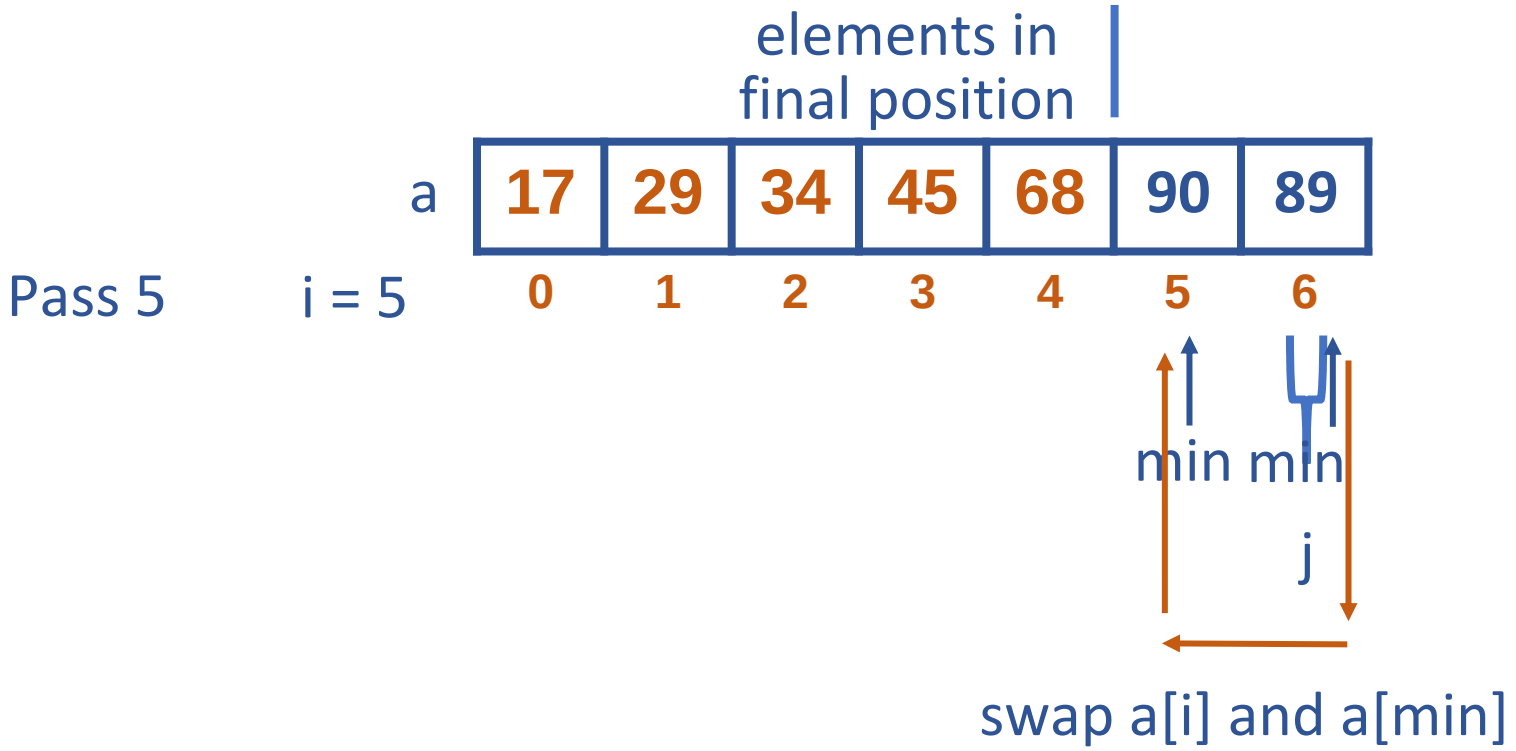


min = 5

Selection Sort

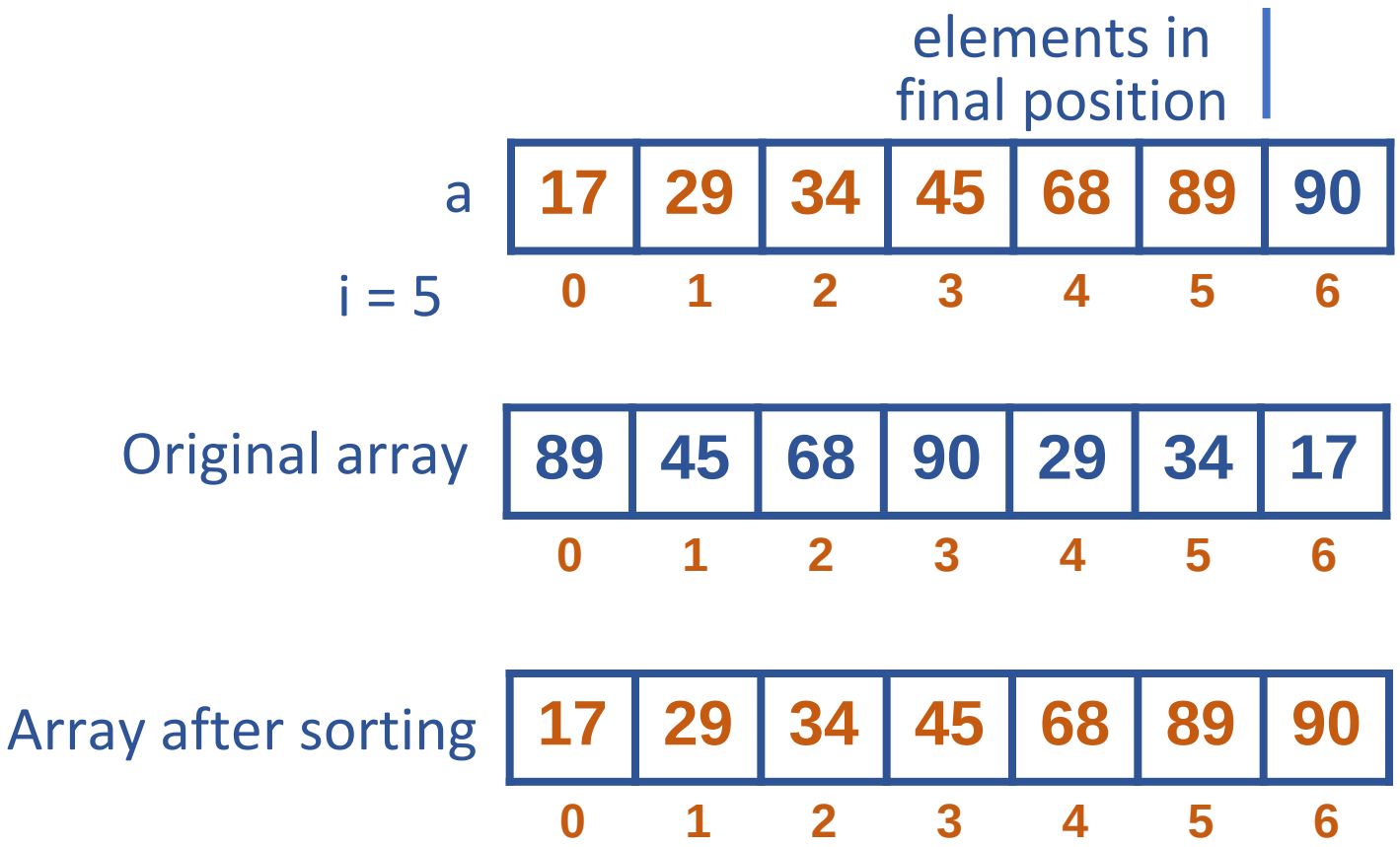


Selection Sort



min = 6

Selection Sort



Selection Sort

```
ALGORITHM  SelectionSort(A[0 .. n -1])
//Sorts a given array by selection sort
//Input: An array A[0 .. n - 1]  of orderable elements
//Output: Array A[0 .. n - 1]  sorted in ascending order
for i <- 0 to n - 2 do
        min <- i
        for j <- i+l to n-1 do
                if A[j] <  A[min]  min  <- j
        swap  A[i] and A[min]
```

Selection Sort

PES UNIVERSITY

Selection Sort Analysis

$$C(n) = \sum_{i=0}^{n-2} \sum_{j=i+1}^{n-1} 1 = \sum_{i=0}^{n-2} (n-1-i) = \frac{(n-1)n}{2}$$

Selection Sort is a $\Theta(n^2)$ algorithm

Bubble Sort

- Compare adjacent elements of the list and exchange them if they are out of order
- By doing it repeatedly, we end up bubbling the largest element to the last position on the list
- The next pass bubbles up the second largest element and so on and after n – 1 passes, the list is sorted
- Pass i ($0 \leq i \leq n-2$) can be represented as follows:

$A[0], A[1], A[2], \ldots, A[j] \overset{?}{\longleftrightarrow} A[j+1], \ldots, A[n-i-1] \mid A[n-i] \leq \ldots \leq A[n-1]$

in their final positions

Bubble Sort

elements in final position

i = 0

89	45	68	90	29	34	17
0	1	2	3	4	5	6

j

j = 0

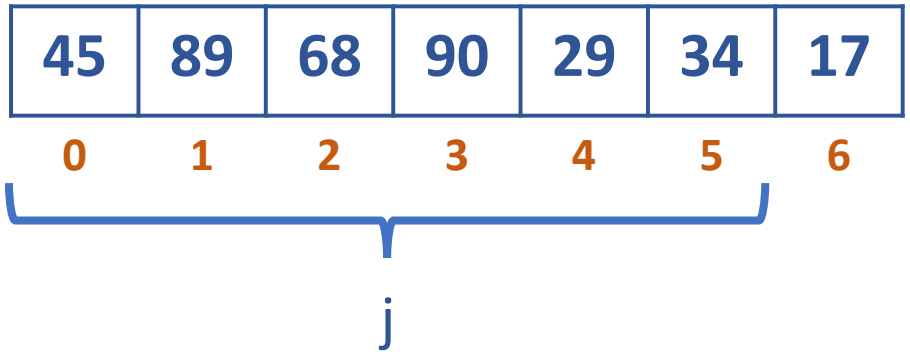
~~89~~ 45	~~45~~ 89	68	90	29	34	17
0	1	2	3	4	5	6

45 < 89 ☑ swap

Bubble Sort

elements in final position

i = 0



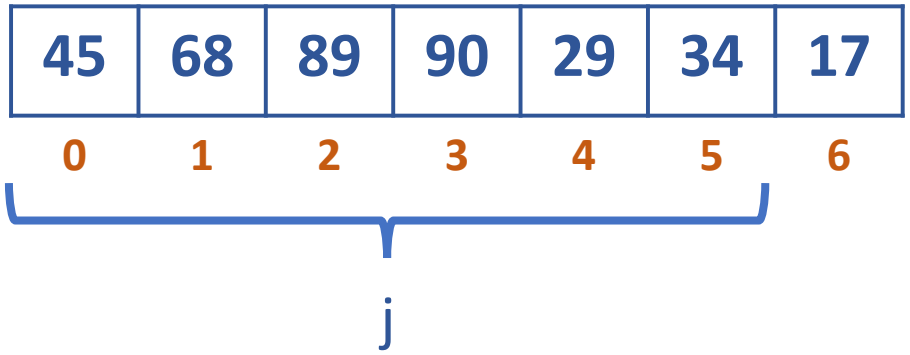
j = 1

68 < 89 ☑ swap

Bubble Sort

elements in final position

i = 0



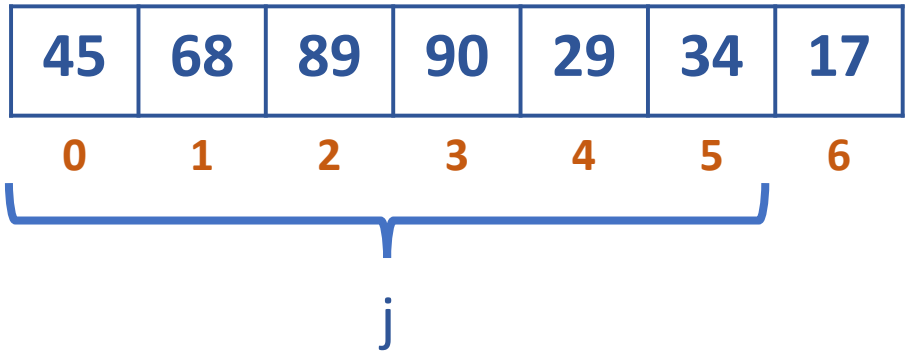
j = 2

90 < 89 ☒ no swap

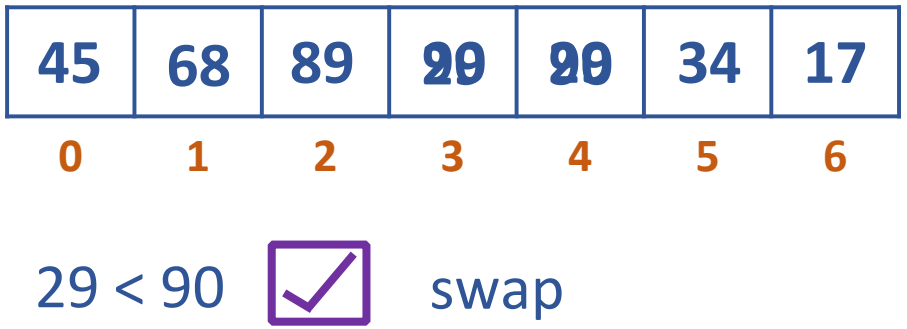
Bubble Sort

elements in final position

i = 0



j = 3

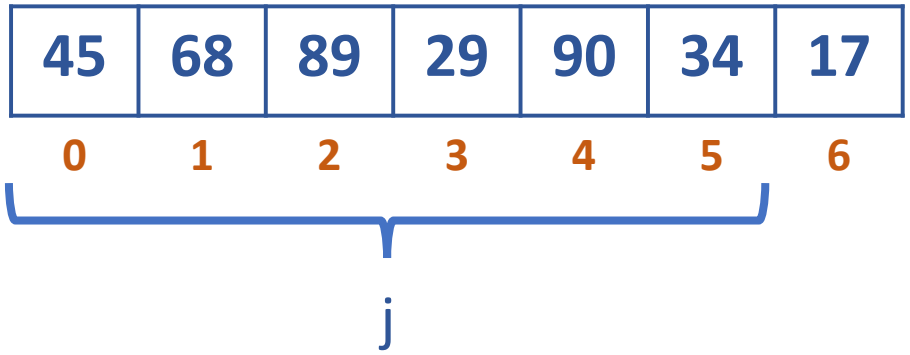


29 < 90 ☑ swap

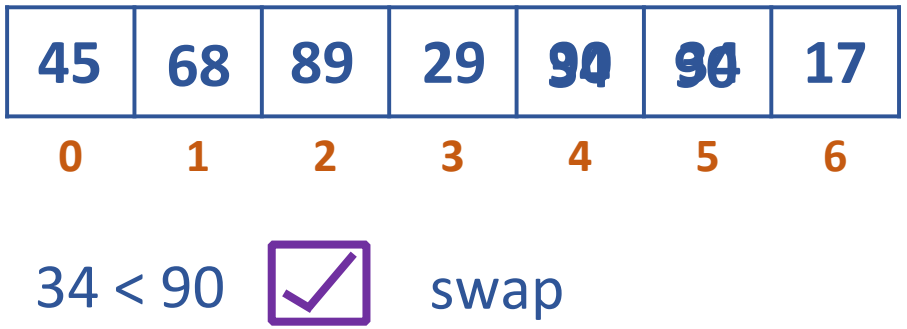
Bubble Sort

elements in final position

i = 0



j = 4

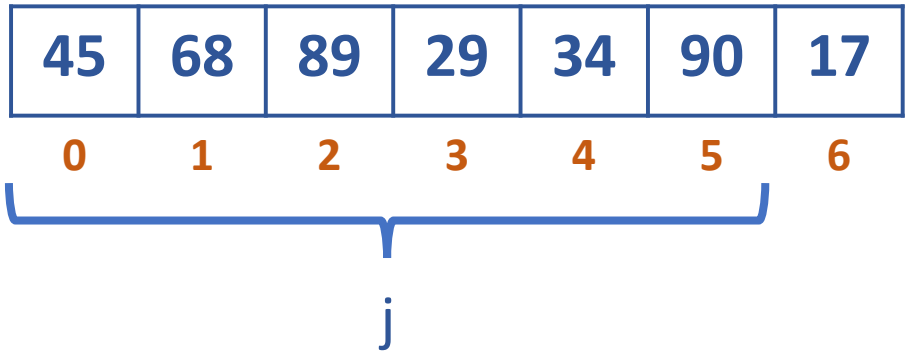


34 < 90 ☑ swap

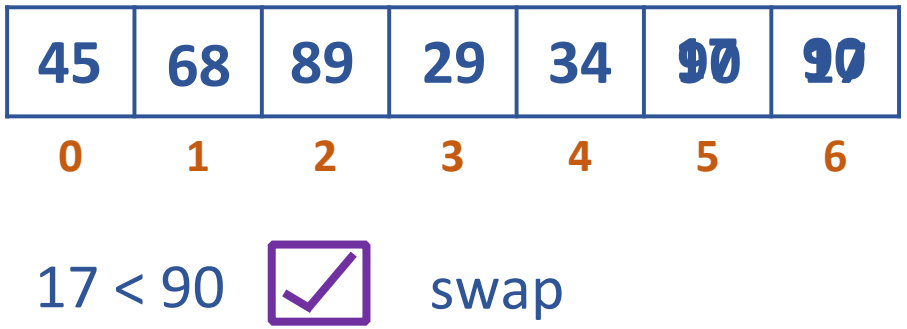
Bubble Sort

elements in final position

i = 0



j = 5



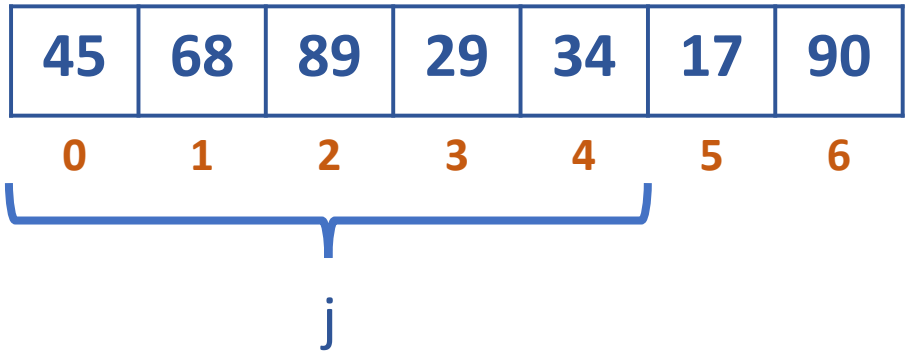
17 < 90 ☑ swap

Bubble Sort

After first iteration

i = 1

elements in final position



j = 0

68 < 45 ☒ no swap

Bubble Sort

elements in final position

i = 1

45	68	89	29	34	17	90
0	1	2	3	4	5	6

j

j = 1

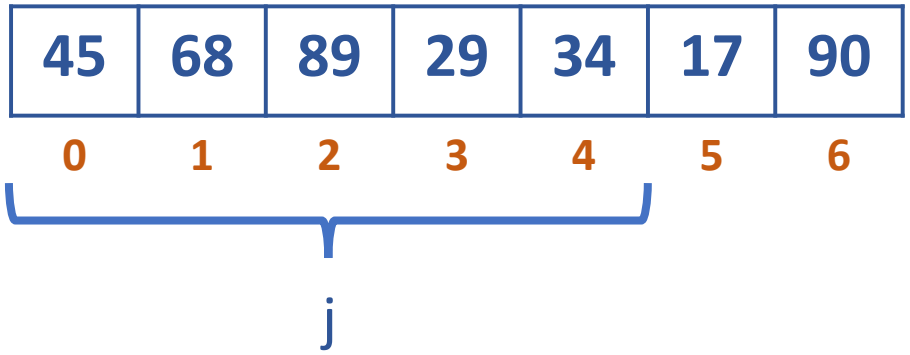
45	68	89	29	34	17	90
0	1	2	3	4	5	6

89 < 68 ☒ no swap

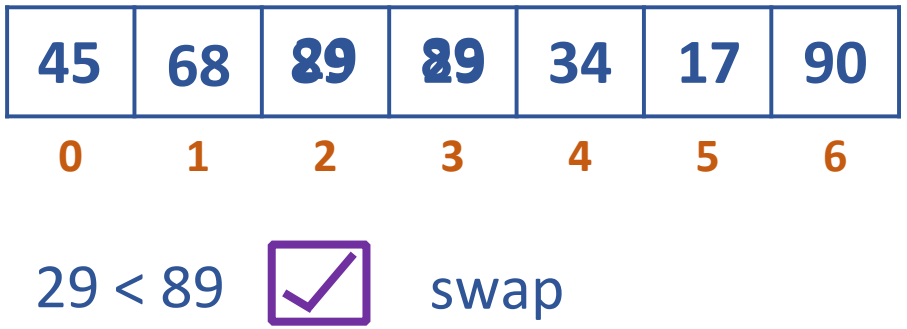
Bubble Sort

elements in final position

i = 1



j = 2

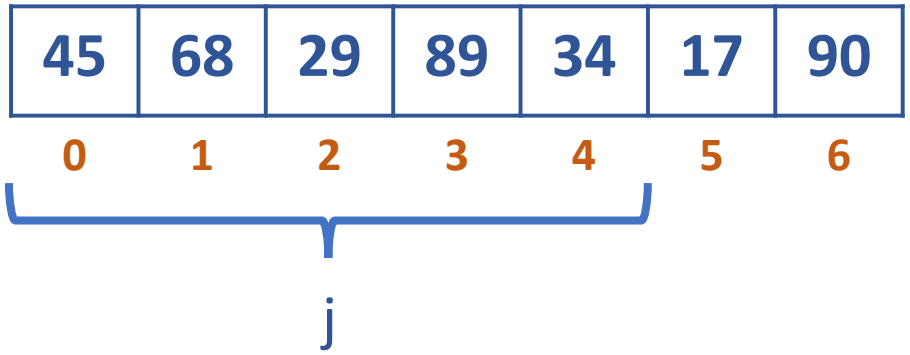


29 < 89 ☑ swap

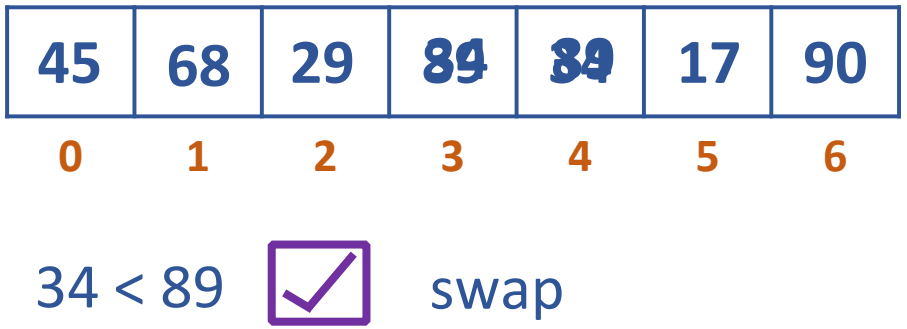
Bubble Sort

elements in final position

i = 1



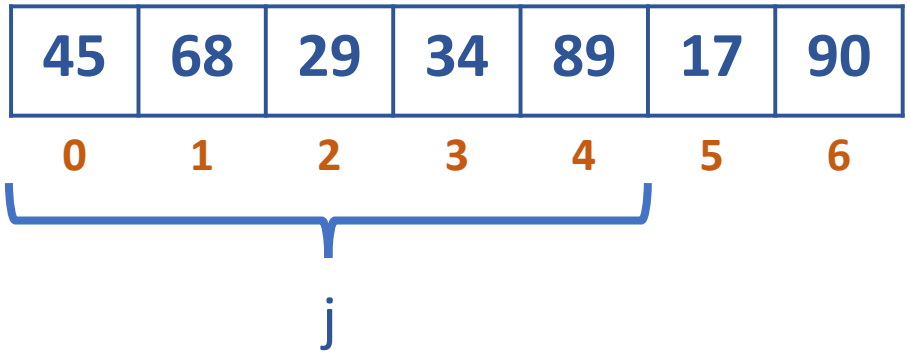
j = 3

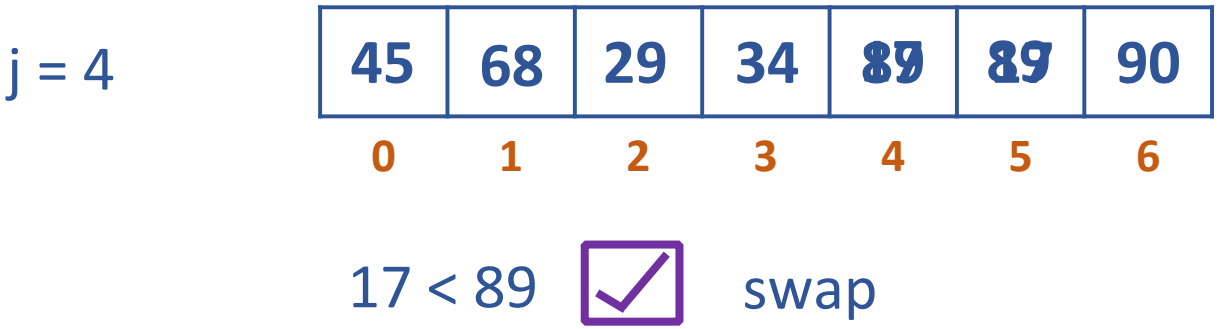


34 < 89 ☑ swap

Bubble Sort

elements in final position



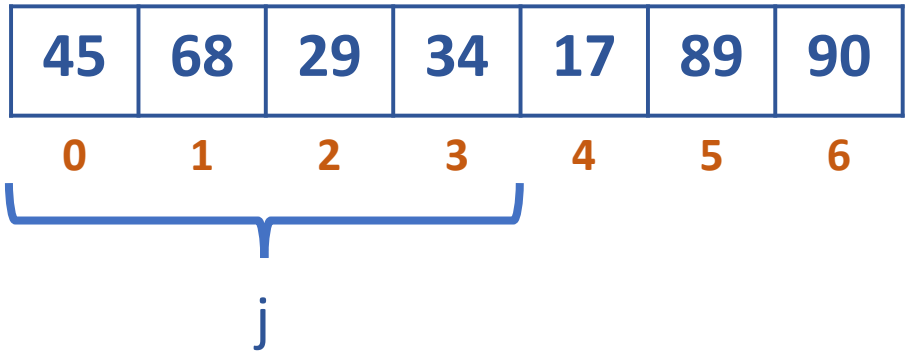


Bubble Sort

After second iteration

i = 2

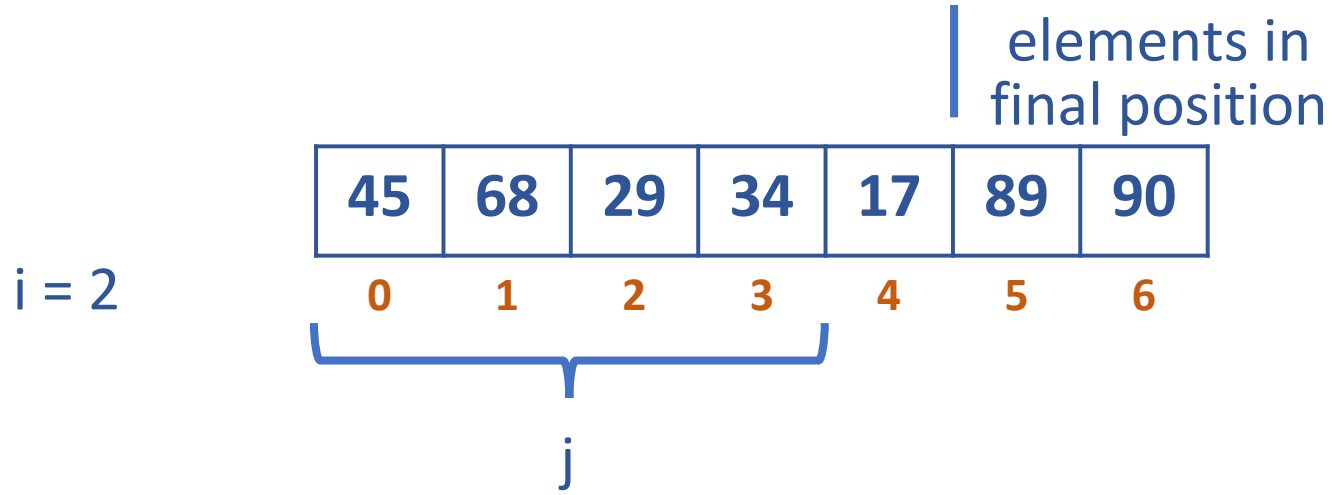
elements in final position

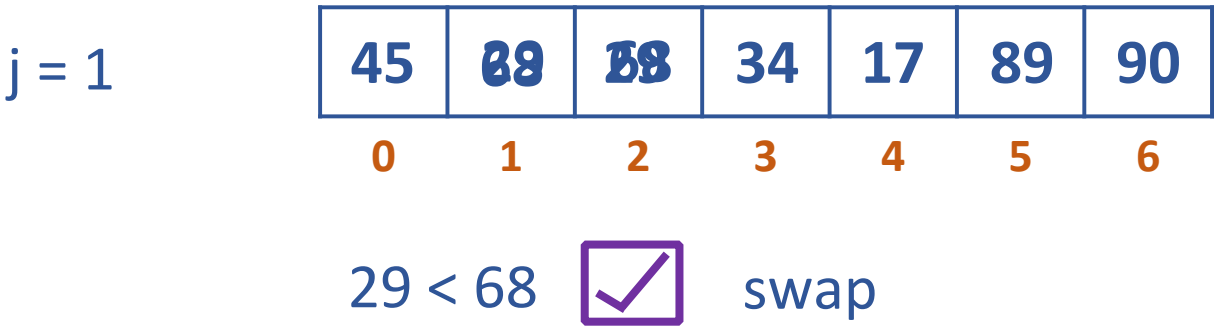


j = 0

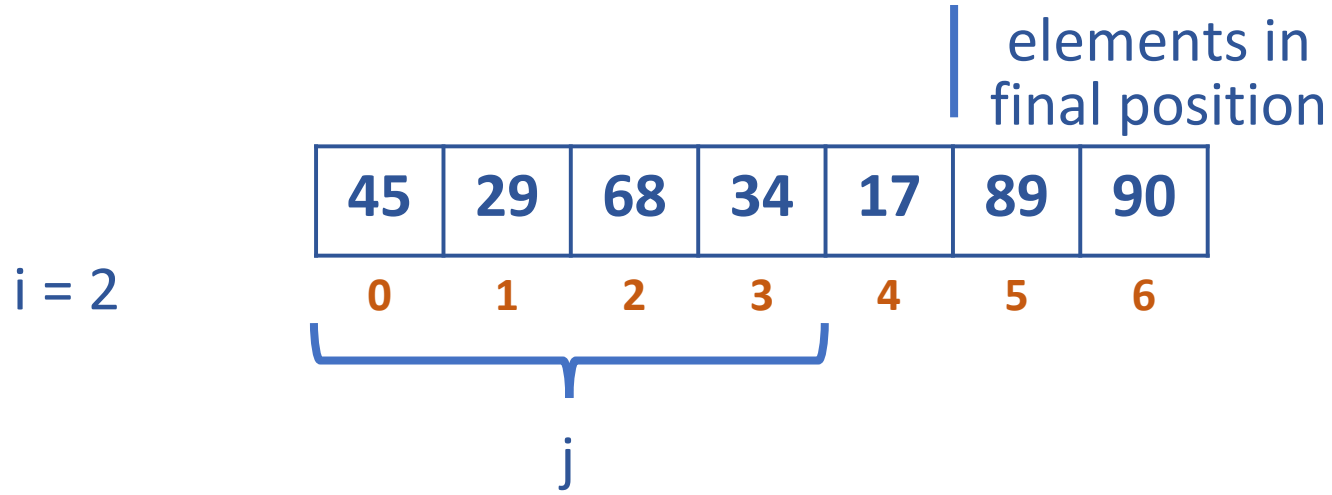
68 < 45 ☒ no swap

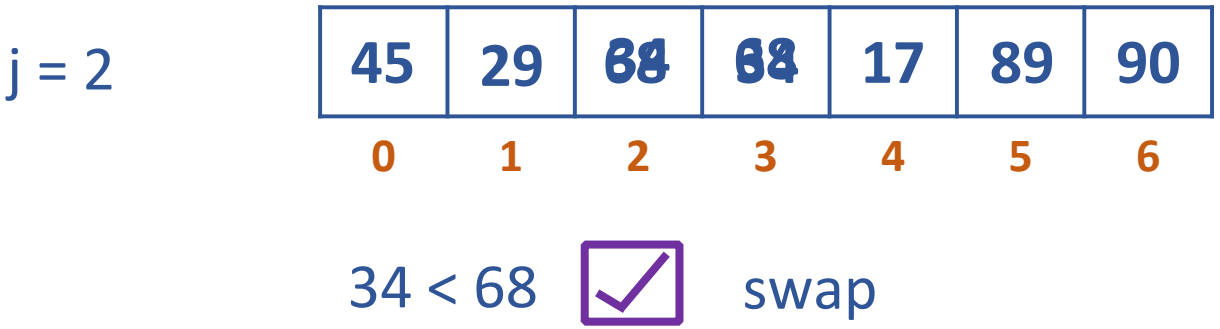
Bubble Sort


elements in final position
45 68 29 34 17 89 90
0 1 2 3 4 5 6
i = 2
j

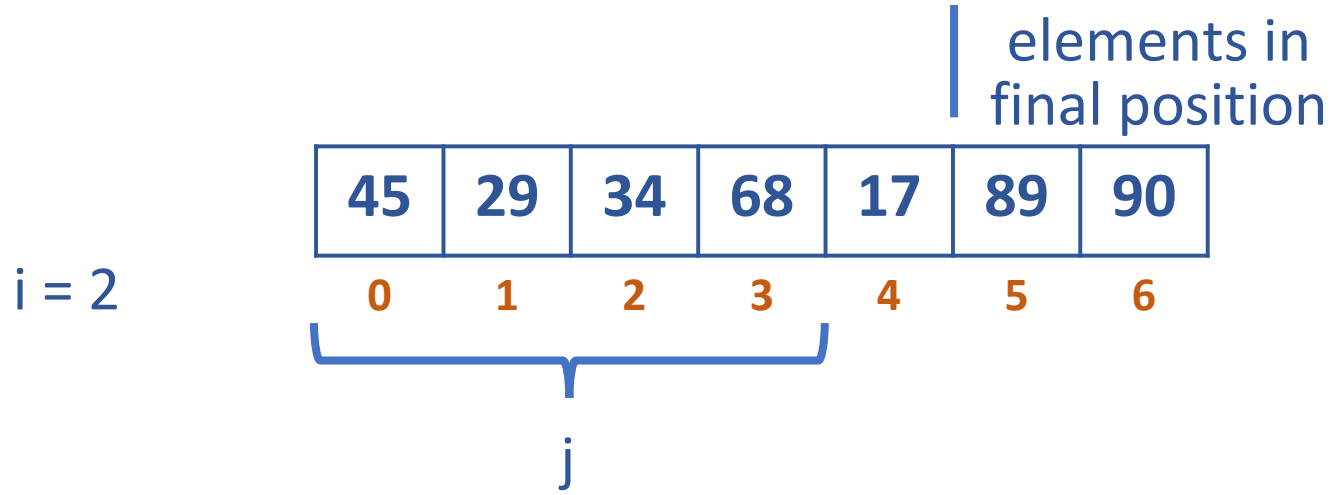

j = 1
45 29 68 34 17 89 90
0 1 2 3 4 5 6
29 < 68
swap

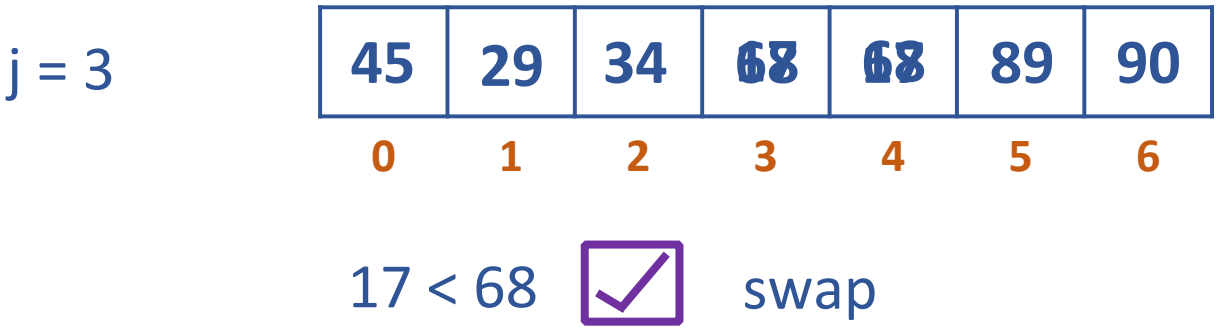
Bubble Sort


elements in
final position
45
29
68
34
17
89
90
0
1
2
3
4
5
6
i = 2
j


j = 2
45
29
34
68
17
89
90
0
1
2
3
4
5
6
34 < 68
swap

Bubble Sort


elements in
final position
45 29 34 68 17 89 90
0 1 2 3 4 5 6
i = 2
j

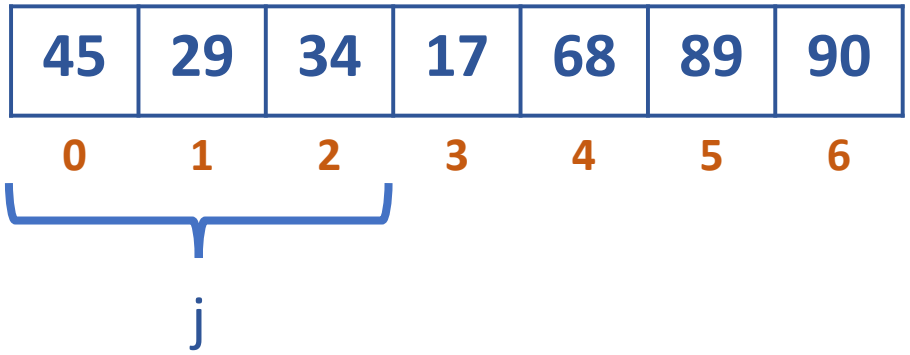

45 29 34 68 68 89 90
0 1 2 3 4 5 6
j = 3
17 < 68
swap

Bubble Sort

After third iteration

i = 3

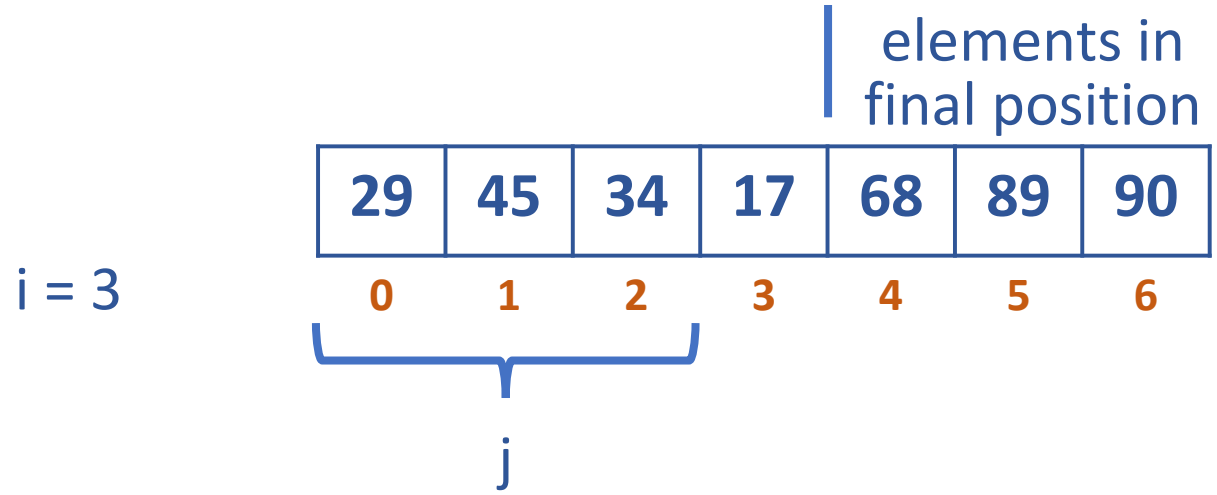
elements in final position

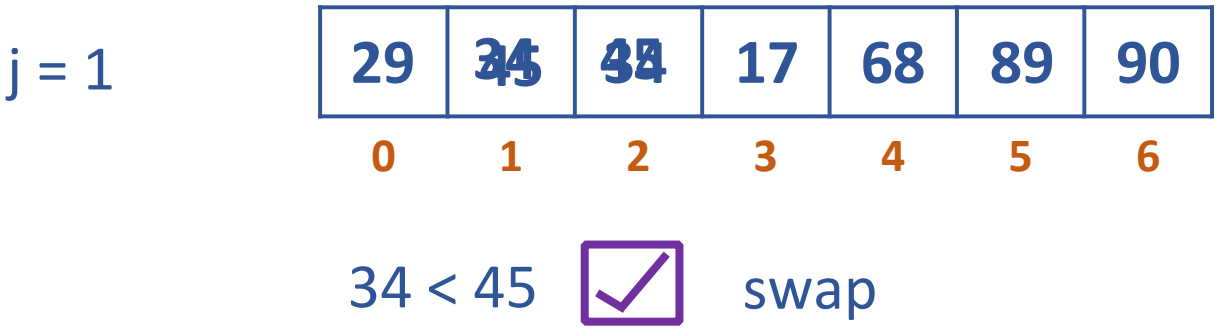


j = 0

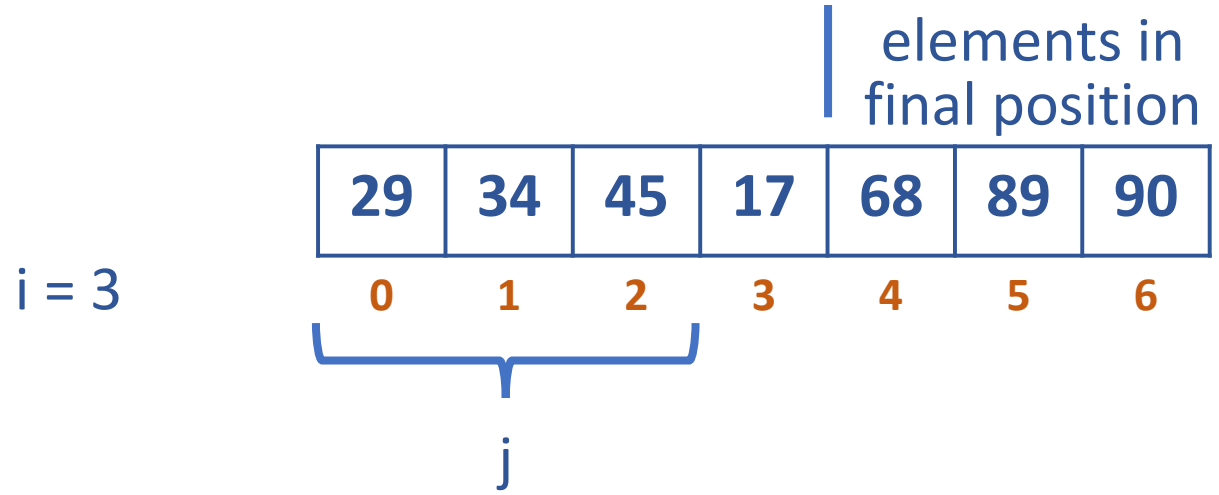
29 < 45 ☑ swap

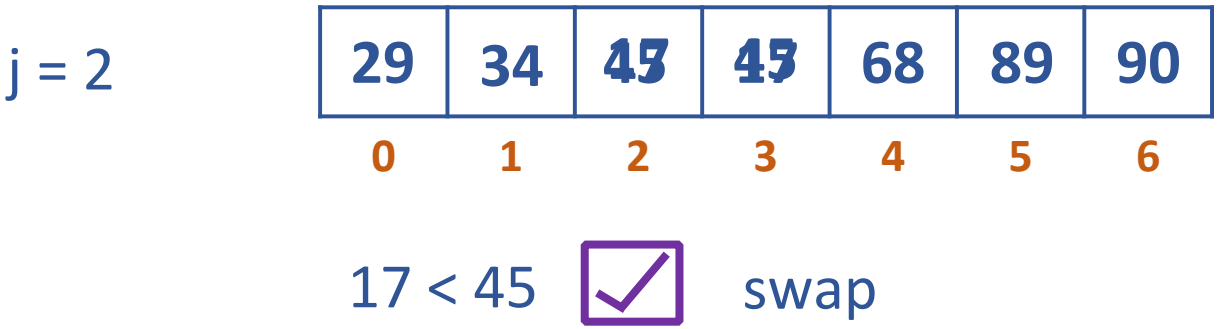
Bubble Sort


elements in
final position
29
45
34
17
68
89
90
0
1
2
3
4
5
6
i = 3
j


j = 1
29
34
45
17
68
89
90
0
1
2
3
4
5
6
34 < 45
swap

Bubble Sort


elements in
final position
29 34 45 17 68 89 90
0 1 2 3 4 5 6
i = 3
j

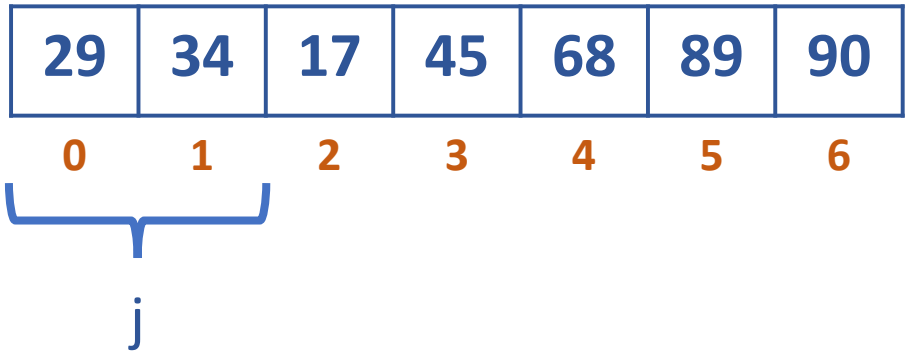

j = 2
29 34 45 17 68 89 90
0 1 2 3 4 5 6
17 < 45
swap

Bubble Sort

After fourth iteration

i = 4

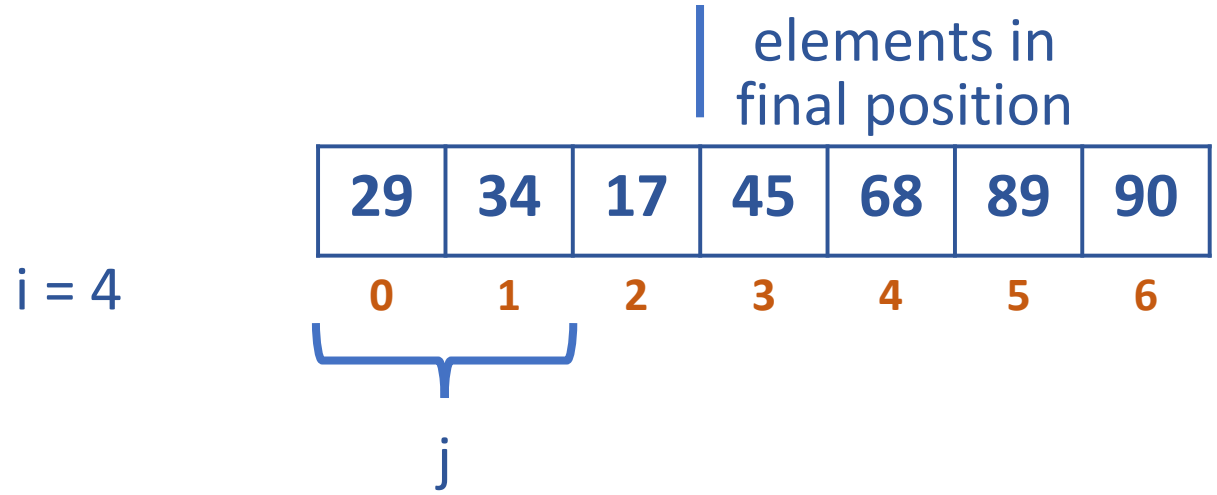
elements in final position

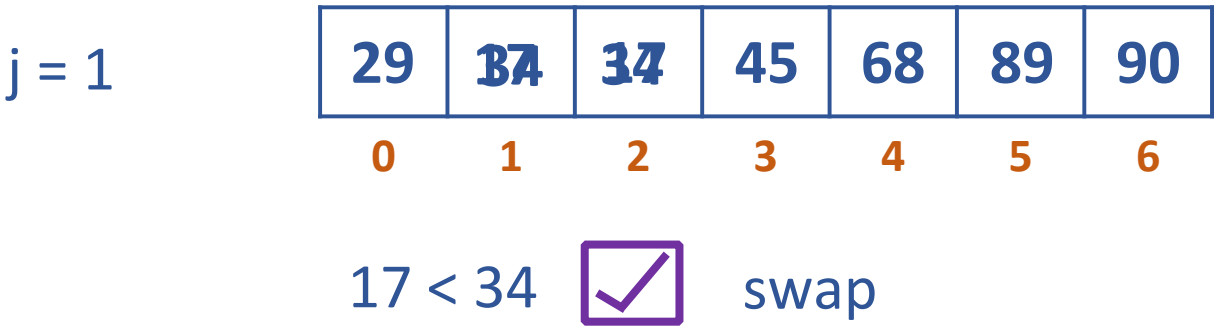


j = 0

34 < 29 ☒ no swap

Bubble Sort


elements in
final position
29
34
17
45
68
89
90
0
1
2
3
4
5
6
i = 4
j

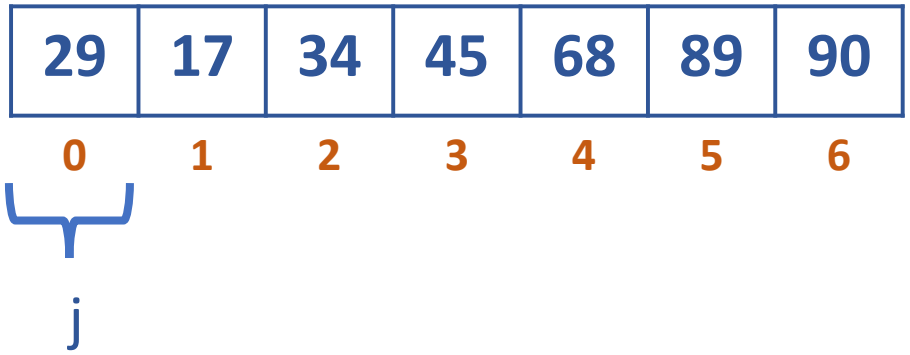

j = 1
29
34
17
45
68
89
90
0
1
2
3
4
5
6
17 < 34
swap

Bubble Sort

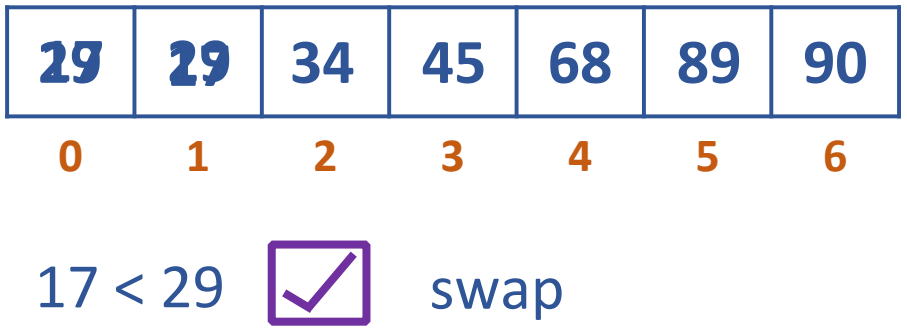
After fifth iteration

i = 5

elements in final position



j = 0



17 < 29 ☑ swap

Bubble Sort

After sixth iteration | elements in final position

17	29	34	45	68	89	90
0	1	2	3	4	5	6

Bubble Sort

```
ALGORITHM  BubbleSort(A[0 .. n - 1])
//Sorts a given array by bubble sort in  their  final  positions
//Input: An array A[0 .. n  - 1]  of orderable elements
//Output: Array A[0 .. n- 1]  sorted in ascending order
for  i  <-- 0 to n - 2 do
   for  j  <-- 0 to n - 2 - i  do
         if A[ j + 1 ]  <  A[ j ]  swap A[ j ] and A[ j + 1 ]
```

Bubble Sort

PES UNIVERSITY

Bubble Sort Analysis

$$C(n) = \sum_{i=0}^{n-2} \sum_{j=0}^{n-2-i} 1 = \sum_{i=0}^{n-2} [(n-2-i) - 0 + 1]$$

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

Bubble Sort is a $\Theta(n^2)$ algorithm

Sequential Search

- Compares successive elements of a given list with a given search key until:
  - A match is encountered (Successful Search)
  - List is exhausted without finding a match (Unsuccessful Search)
- An improvisation to the algorithm is to append the key to the end of the list
- This means the search has to be successful always and we can eliminate the end of list check

Sequential Search

- Sequential / Linear Search

10	14	19	26	27	31	33	35	42	44
0	1	2	3	4	5	6	7	8	9

- For key = 33, 6 is returned
- For key = 50, -1 is returned

Sequential Search

```
AlGORITHM  SequentialSearch2(A[0 .. n ],  K)
//Implements sequential search with a search key as  a sentinel
//Input: An array A of n elements and a search key  K
//Output: The index of the first element in A[0 .. n -1] whose value is
//  equal to K or -1 if no such element is found
A[n]<---K
i<---0
while A[i] ≠K  do
   i<--- i + 1
if i  <  n return i
else return -1
```

Sequential Search

Sequential Search Analysis

- Sequential Search is a Θ(n) algorithm

Brute – Force String Matching

String Matching - Terms

- pattern:

  a string of m characters to search for

- text:

  a (longer) string of n characters to search in

- problem:

  find a substring in the text that matches the pattern

DESIGN AND ANALYSIS OF ALGORITHMS

String Matching Idea

Step 1: Align pattern at beginning of text

Step 2: Moving from left to right, compare each character of pattern to the corresponding character in text until:

- all characters are found to match (successful search); or
- a mismatch is detected

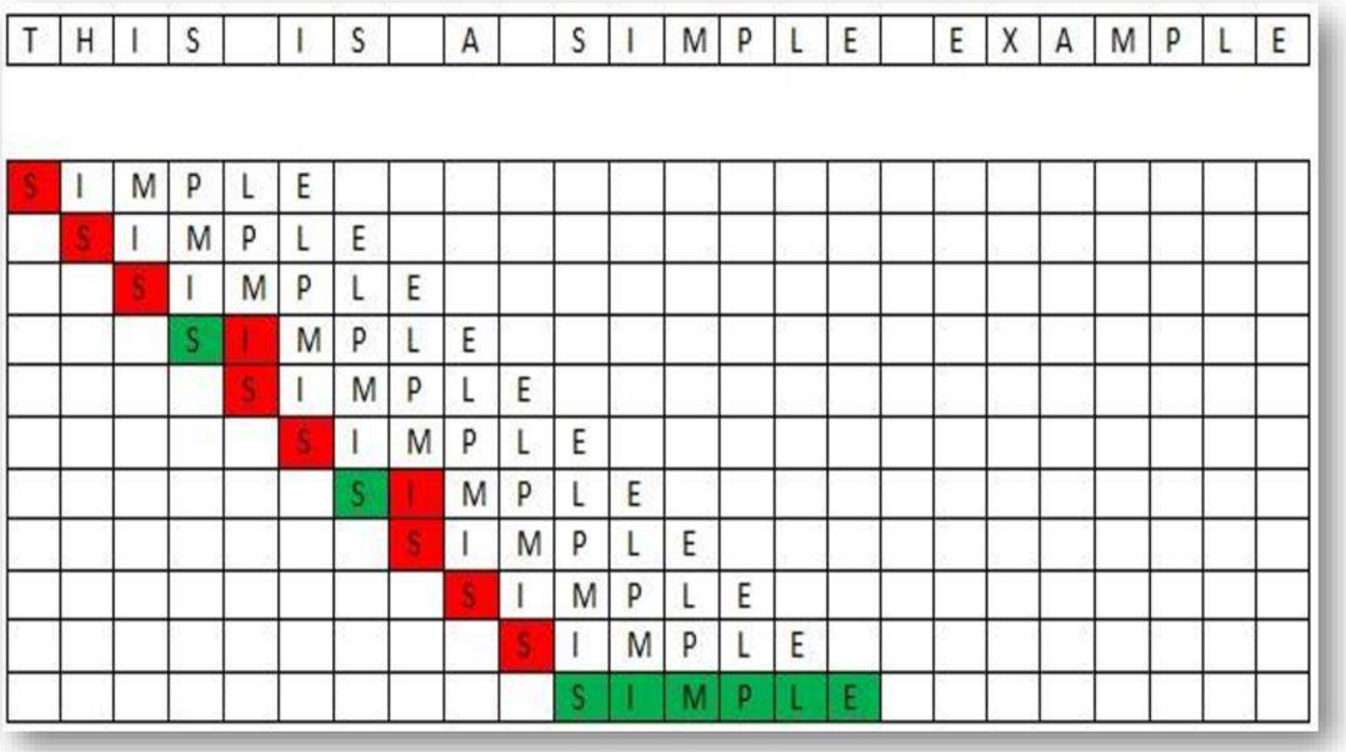
Step 3: While pattern is not found and the text is not yet exhausted, realign pattern one position to the right and repeat Step 2

String Matching

```
ALGORITHM  BruteForceStringMatch(T[0 .. n -1], P[0 .. m -1])
//Implements brute-force string matching
//Input: An array T[0 .. n  - 1]  of n  characters representing a text
// and an array  P[0 .. m - 1] of m characters representing a pattern
//Output: The index of the first character in the text that starts a
//matching substring or -1 if the search is unsuccessful
for i ← 0 to n-m do
   j ← 0
   while  j  <  m  and P[j] =  T[i + j] do
          j ←j+1
   if j  =  m  return i
return -1
```

DESIGN AND ANALYSIS OF ALGORITHMS

String Matching Example



String Matching Analysis

Worst Case:

- The algorithm might have to make all the 'm' comparisons for each of the (n-m+1) tries
- Therefore, the algorithm makes m(n-m+1) comparisons
- Brute Force String Matching is a O(nm) algorithm

DESIGN AND ANALYSIS OF ALGORITHMS

Exhaustive Search

- Exhaustive Search is a brute – force problem solving technique
- It suggests generating each and every element of the problem domain, selecting those of them that satisfy all the constraints and then finding a desired element
- The desired element might be one which minimizes or maximizes a certain characteristic
- Typically the problem domain involves combinatorial objects such as permutations, combinations and subsets of a given set

Exhaustive Search - Method

- Generate a list of all potential solutions to the problem in a systematic manner
- Evaluate potential solutions one by one, disqualifying infeasible ones and, for an optimization problem, keeping track of the best one found so far
- When search ends, announce the solution(s) found

Travelling Salesman Problem

- Given a list of cities and the distances between each pair of cities, what is the shortest possible route that visits each city and returns to the origin city?

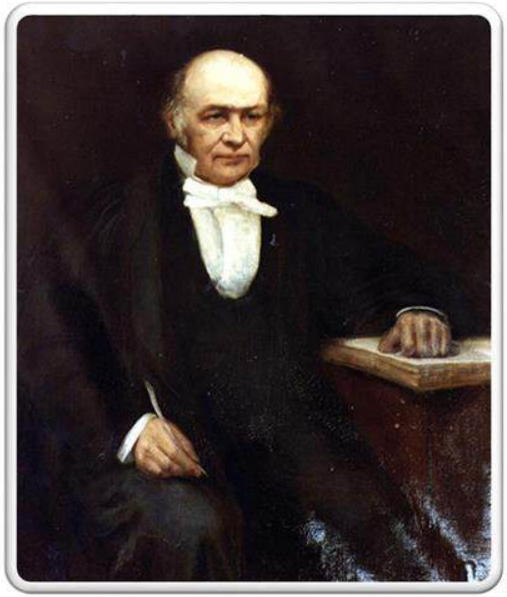
Alternative way to state the problem:

- Find the shortest Hamiltonian Circuit in a weighted connected graph

DESIGN AND ANALYSIS OF ALGORITHMS

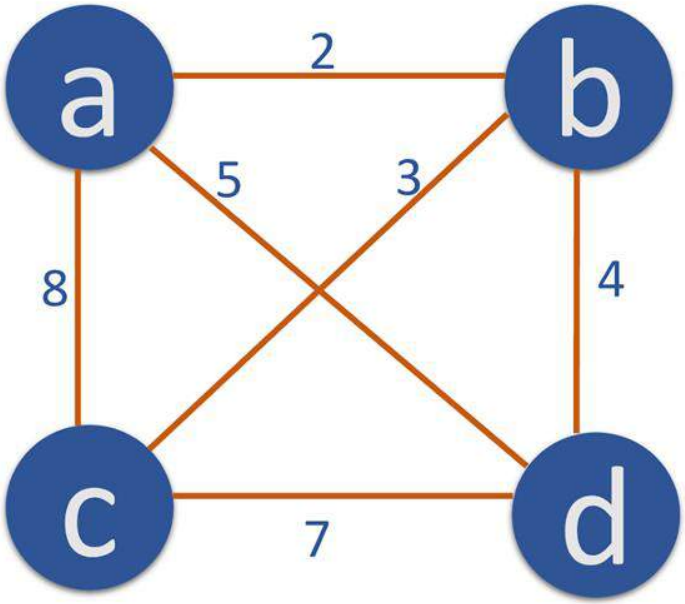
TSP: History and Relevance

- The Travelling Salesman Problem was mathematically formulated by Irish Mathematician Sir William Rowan Hamilton
- It is one of the most intensively studied problems in optimization
- It has applications in logistics and planning



Travelling Salesman Problem

Example



Tour	Length
a→b→c→d→a	**2+3+7+5 = 17**
a→b→d→c→a	2+4+7+8 = 21
a→c→b→d→a	8+3+4+5 = 20
a→c→d→b→a	8+7+4+2 = 21
a→d→b→c→a	5+4+3+8 = 20
a→d→c→b→a	**5+7+3+2 = 17**

Travelling Salesman Problem

Efficiency

- The Exhaustive Search solution to the Travelling Salesman problem can be obtained by keeping the origin city constant and generating permutations of all the other n – 1 cities
- Thus, the total number of permutations needed will be (n – 1)!

Knapsack Problem

Given n items:

- weights: w_1 w_2 ... w_n
- values: v_1 v_2 ... v_n
- a knapsack of capacity W

Find the most valuable subset of items that fit into the knapsack

Knapsack Problem

Example

Knapsack Capacity W = 16

Item	Weight	Value
1	2	20
2	5	30
3	10	50
4	5	10

Knapsack Problem

Subset	Total Weight	Total Value
{1}	2	20
{2}	5	30
{3}	10	50
{4}	5	10
{1, 2}	7	50
{1, 3}	12	70
{1, 4}	7	30
{2, 3}	15	80
{2, 4}	10	40
{3, 4}	15	60
{1, 2, 3}	17	Not Feasible
{1, 2, 4}	12	60
{1, 3, 4}	17	Not Feasible
{2, 3, 4}	20	Not Feasible
{1, 2, 3, 4}	22	Not Feasible

Knapsack Problem by Exhaustive Search

Knapsack Problem

- The Exhaustive Search solution to the Knapsack Problem is obtained by generating all subsets of the set of n items given and computing the total weight of each subset in order to identify the feasible subsets
- The number of subsets for a set of n elements is 2^n
- The Exhaustive Search solution to the Knapsack Problem belongs to $\Omega(2^n)$

The Assignment Problem

- There are n people who need to be assigned to n jobs, one person per job. The cost of assigning person i to job j is C[i, j]. Find an assignment that minimizes the total cost

The Assignment Problem

Example

	Job 1	Job 2	Job 3	Job 4
Person 1	9	2	7	8
Person 2	6	4	3	7
Person 3	5	8	1	8
Person 4	7	6	9	4

The Assignment Problem

Algorithmic Plan

1. Generate all legitimate assignments
2. Compute their costs
3. Select the cheapest one

The Assignment Problem

The Assignment Problem by Exhaustive Search

Assignment	Cost
1, 2, 3, 4	9 + 4 + 1 + 4 = 18
1, 3, 4, 2	9 + 8 + 9 + 7 = 33
1, 4, 3, 2	9 + 6 + 1 + 7 = 23
1, 4, 2, 3	9 + 6 + 3 + 8 = 26
1, 3, 2, 4	9 + 8 + 3 + 4 = 24
1, 2, 4, 3	9 + 4 + 9 + 8 = 30

etc.,

The Assignment Problem

Efficiency

- The Assignment Problem is solved by generating all permutations of n
- The number of permutations for a given number n is n!
- Therefore, the exhaustive search is impractical for all but very small instances of the problem

Decrease and Conquer: Binary Search

Decrease-by-a-Constant-Factor Algorithms:

```
Algorithm BinarySearchRec(A[0..n-1], K)
      if(n <= 0)
            return -1
      m = ⌊n/2⌋
      if(k = A[m])
            return m
      if(k < A[m])
            return BinarySearchRec(A[0..m-1], K)
      else
            return BinarySearchRec(A[m+1..n-1], K)
```

Worst Case Time complexity: $O(Log_2 n)$

Fake-Coin Problem

Among n identical-looking coins, one is fake.With a balance scale, we can compare any two sets of coins.

The problem is to design an efficient algorithm for detecting the fake coin.

The most natural idea for solving this problem is to divide n coins into two piles of $n/2$ coins each, leaving one extra coin aside if n is odd, and put the two piles on the scale.

If the piles weigh the same, the coin put aside must be fake; otherwise, we can proceed in the same manner with the lighter pile, which must be the one with the fake coin.

$W(n) = W(n/2) + 1$ for $n > 1$, $W(1) = 0$.

$W(n) = \log_2 n$.

It would be more efficient to divide the coins not into two but into *three* piles of about $n/3$ coins each.

Russian Peasant Multiplication

Let n and m be positive integers whose product we want to compute, and let us measure the instance size by the value of n.
if n is even, an instance of half the size has to deal with $n/2$, and we have an obvious formula relating the solution to the problem's larger instance to the solution to the smaller one:
$n \cdot m = (n/2) * 2m$

If n is odd, we need only a slight adjustment of this formula:
$n \cdot m = ((n-1)/2) \cdot 2m + m$.
Using these formulas and the trivial case of $1 \cdot m = m$ to stop, we can compute product $n \cdot m$ either recursively or iteratively.

n	m	
50	65	
25	130	
12	260	(+130)
6	520	
3	1040	
1	2080	(+1040)
	2080	+(130 + 1040) = 3250

(a)

n	m	
50	65	
25	130	130
12	260	
6	520	
3	1040	1040
1	2080	2080
		3250

(b)

FIGURE 4.11 Computing 50 · 65 by the Russian peasant method.

Josephus problem

Named for Flavius Josephus, a famous Jewish historian who participated in and chronicled the Jewish revolt of 66–70 c.e. against the Romans.
Josephus, as a general, managed to hold the fortress of Jotapata for 47 days, but after the fall of the city he took refuge with 40 diehards in a nearby cave. There, the rebels voted to perish rather than surrender. Josephus proposed that each man in turn should dispatch his neighbor, the order to be determined by casting lots. Josephus contrived to draw the last lot, and, as one of the two surviving men in the cave, he prevailed upon his intended victim to surrender to the Romans.

Josephus Problem

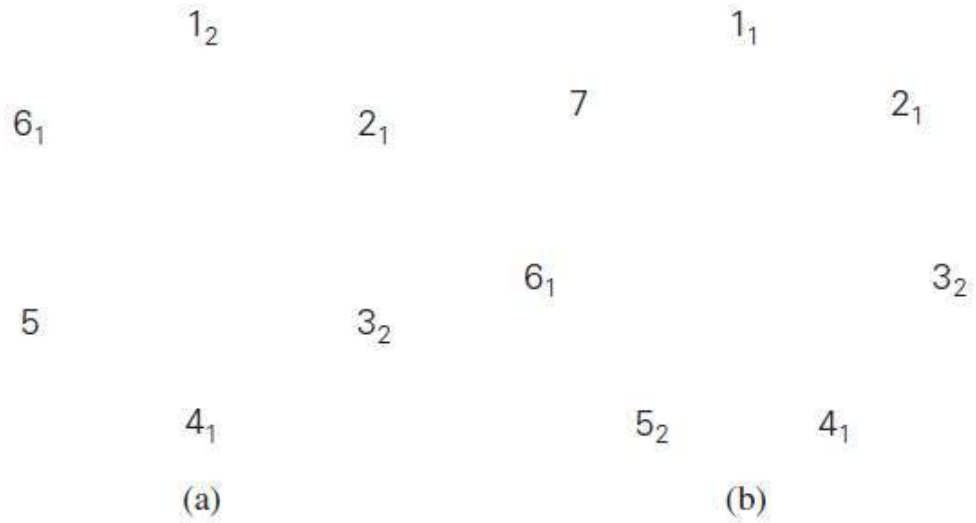
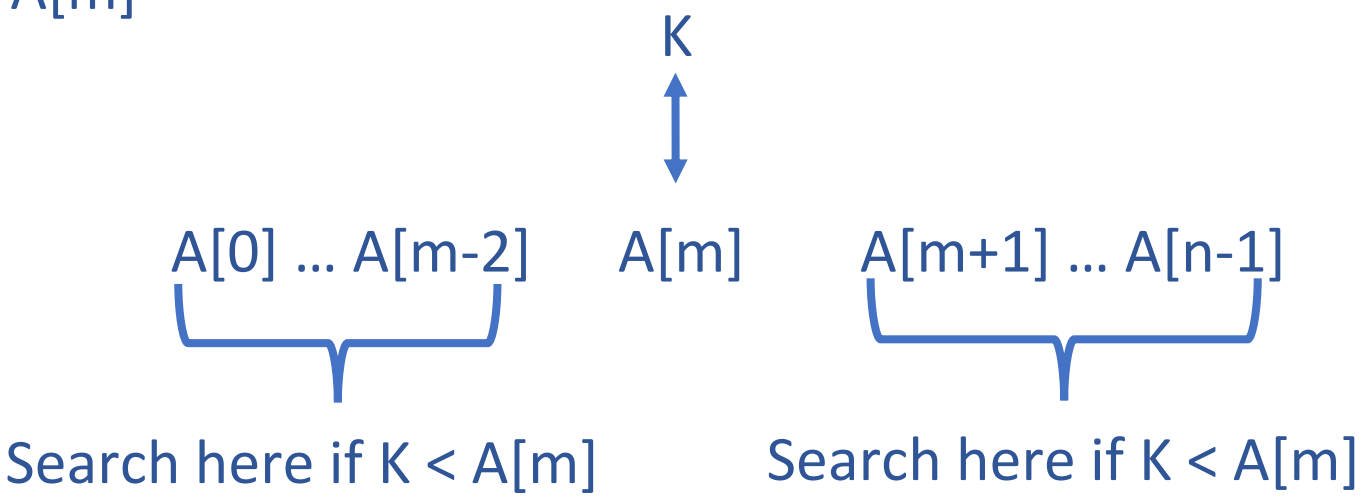


FIGURE 4.12 Instances of the Josephus problem for (a) $n = 6$ and (b) $n = 7$. Subscript numbers indicate the pass on which the person in that position is eliminated. The solutions are $J(6) = 5$ and $J(7) = 7$, respectively.

Binary Search - Idea

- Binary Search is a remarkably efficient algorithm for searching in a sorted array
- It works by comparing the search key K with the array's middle element A[m]
- If they match, the algorithm stops
- Otherwise, the same operation is repeated recursively for the first half of the array if K < A[m] and for the second half if K > A[m]



Binary Search - Algorithm

```
ALGORITHM  BinarySearch(A[0 .. n  -1], K)
// Implements non recursive 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
```

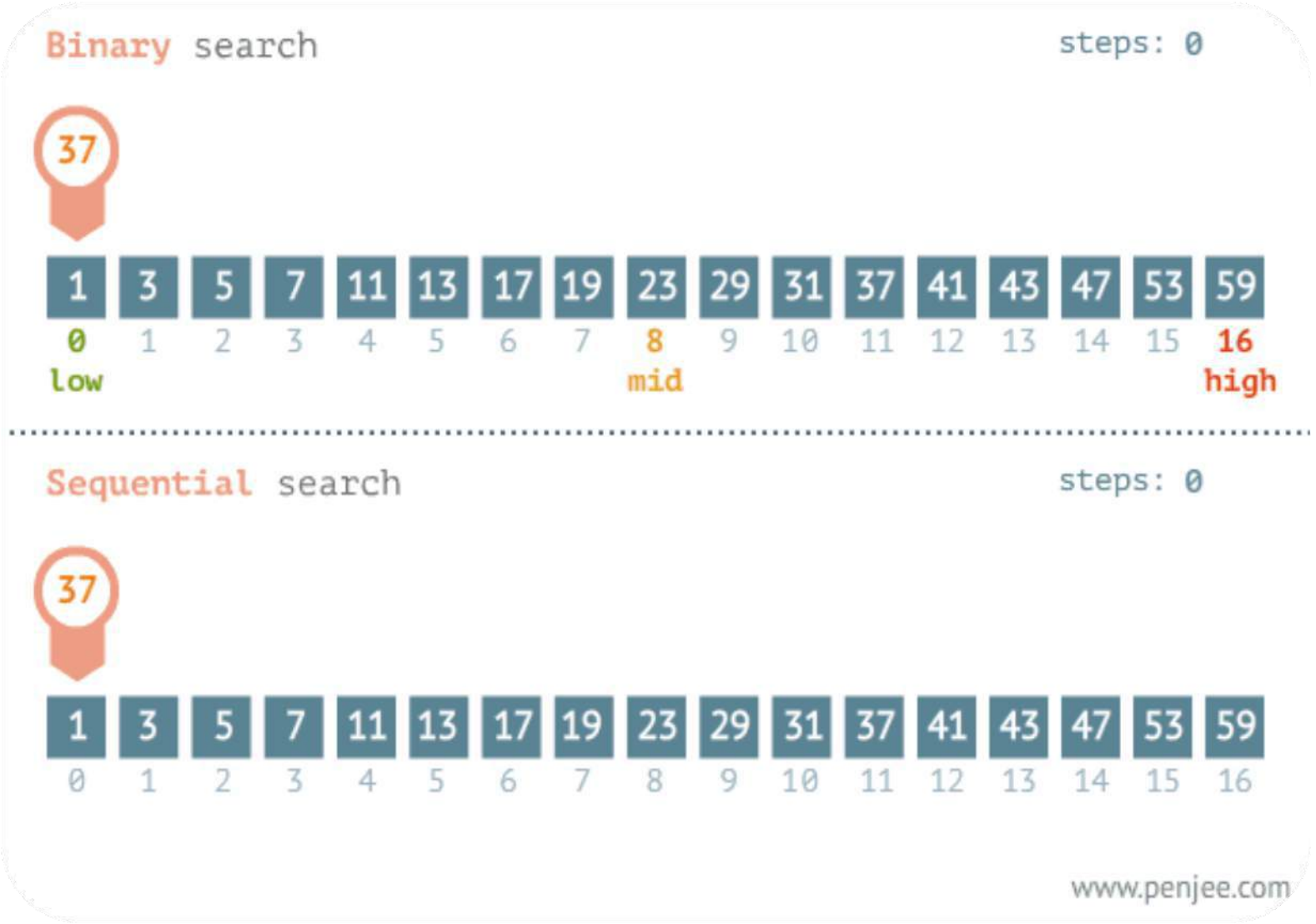
Binary Search - Example

Search Key K = 70

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*				

DESIGN AND ANALYSIS OF ALGORITHMS

Binary Search Vs Linear Search



Binary Search - Analysis: Worst Case

The basic operation is the comparison of the search key with an element of the array

The number of comparisons made are given by the following recurrence:

$$C_{worst}(n) = C_{worst}\left(\left\lfloor n/2 \right\rfloor\right) + 1 \; for \; n > 1, C_{worst}(1) = 1$$

For the initial condition $C_{worst}(1) = 1$, we obtain:

$$C_{worst}(2^k) = k + 1 = \log_2 n + 1$$

For any arbitrary positive integer, n:

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

Binary Search - Analysis: Average Case

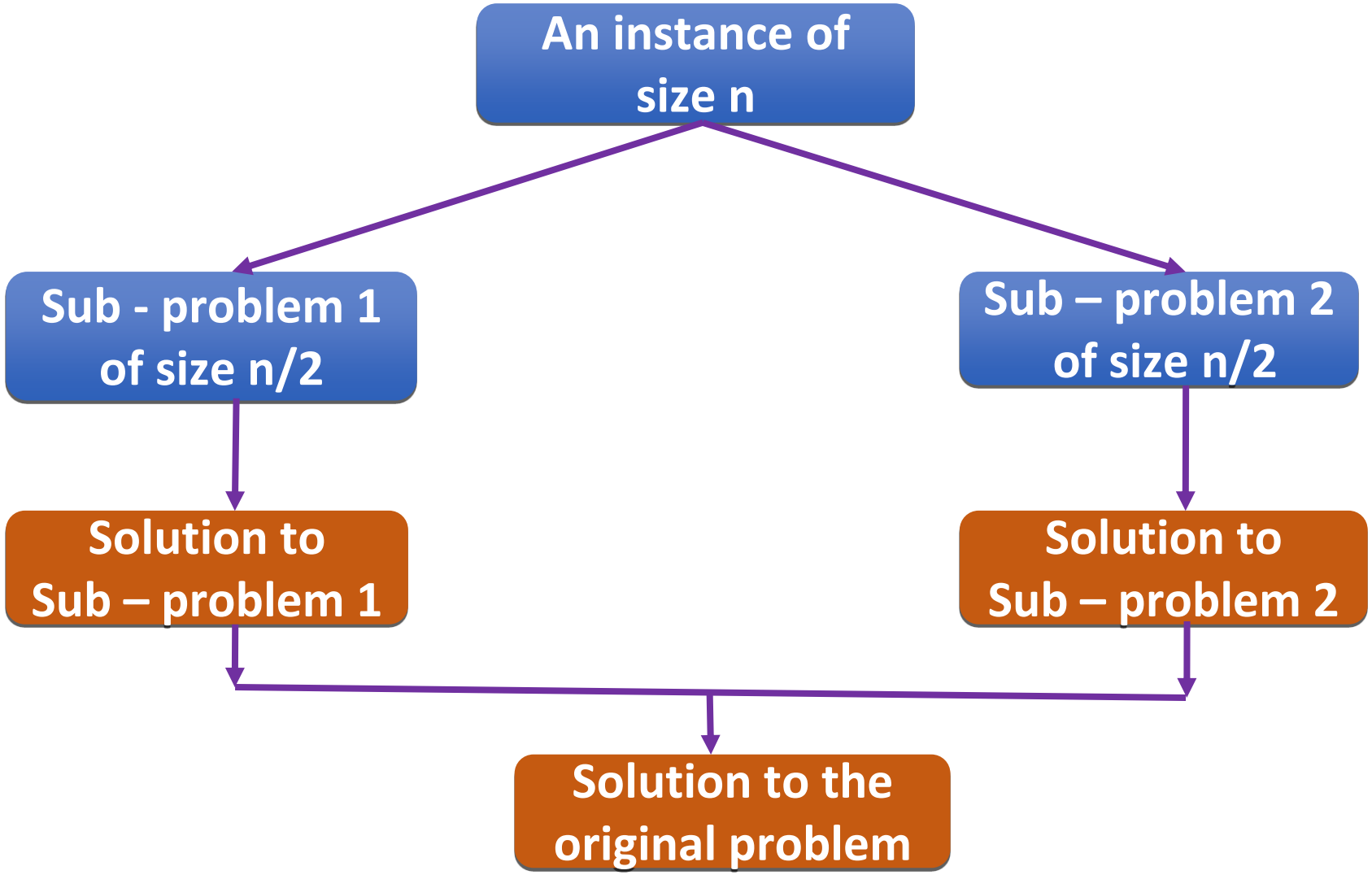
$$C_{avg} \approx \log_2 n$$

Divide and Conquer – Idea

- Divide and Conquer is one of the most well – known algorithm design strategies
- The principle underlying Divide and Conquer strategy can be stated as follows:
  - Divide the given instance of the problem into two or more smaller instances
  - Solve the smaller instances recursively
  - Combine the solutions of the smaller instances and obtain the solution for the original instance

Divide and Conquer – Idea

- Divide and Conquer



General Divide and Conquer

Recurrence

- In the most typical cases of Divide and Conquer, a problem's instance of size n can be divided into b instances of size n/b, with a of them needing to be solved
- Here a and b are constants; a >= 1 and b >= 1
- Assuming that size n is a power of b, we get the following recurrence for the running time:

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

- f(n) is a function that accounts for the time spent on dividing the problem and combining the solutions

Master Theorem

Recurrence

- For the recurrence:

$$\mathbf{T(n) = a * T(n/b) + f(n)}$$

- If $f(n) \in \Theta(n^d)$, where d >= 0 in the recurrence relation, 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 O and Ω as well!

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 - Algorithm

```
ALGORITHM  Quicksort(A[l .. r])
// Sorts a subarray by quicksort
// Input: A subarray A[l … r] of A[0 .. n  -1], defined by its left and
// right indices l  and r
// Output: Subarray A[l .. r]  sorted in non decreasing order
if l  <  r
   s ←Partition(A[l .. r])          //s is a split position
   Quicksort(A[l  .. s - 1])
   Quicksort(A[s + 1 .. r])
```

Quick Sort - Algorithm

```
ALGORITHM  Partition(A[l .. r])
// Partitions a subarray by using its first  element as  a pivot
// Input: A  subarray A[l..r] of A[0 .. n  - 1],  defined by its left and right  indices l and r  (l  <  r)
// Output: A 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[i],  A[j])
return  j
```

Quick Sort - Example

5 3 1 9 8 2 4 7

2 3 1 4 5 8 9 7

1 2 3 4 5 7 8 9

1 2 3 4 5 7 8 9

1 2 3 4 5 7 8 9

1 2 3 4 5 7 8 9

Quick Sort - Analysis: Best Case

- The number of comparisons in the best case satisfies the recurrence:
- $C_{best}(n) = 2C_{best}(n/2) + n$ for $n > 1$, $C_{best}(1) = 0$

- According to Master Theorem

$$C_{best}(n) \in \Theta(n \log_2 n)$$

Quick Sort - Analysis: Worst Case

- The number of comparisons in the worst case satisfies the recurrence

$$C_{worst}(n) = (n+1) + n + \cdots + 3 = \frac{(n+1)(n+2)}{2} - 3 \in \theta(n^2)$$

Quick Sort - Analysis: Average Case

Let $C_{avg}(n)$ be the number of key comparisons made by Quick Sort on a randomly ordered array of size n

$$C_{avg}(n) = \frac{1}{n}\sum_{s=0}^{n-1}[(n+1) + C_{avg}(s) + C_{avg}(n-1-s)] \quad \text{for n>1}$$

The solution for the above recurrence is:

$$C_{avg}(n) \approx 2n \ln n \approx 1.38n \log_2 n$$

The third principal variety of decrease-and-conquer, the **size reduction pattern** varies from one iteration of the algorithm to another.

This algorithm helps optimize:

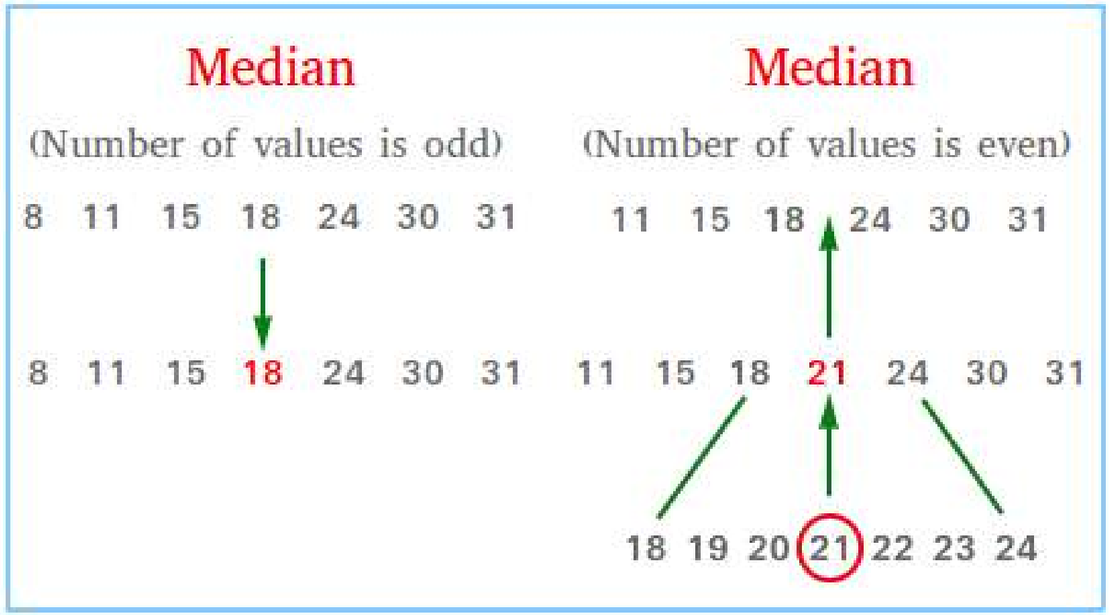
1. Computing a Median and the Selection Problem
2. Interpolation Search
3. Searching and Insertion in a Binary Search Tree
4. The Game of Nim

We will look at **Computing a Median and the Selection Problem** in detail.

Computing a Median and the Selection Problem

Objective: Find the median of an odd array in the best time.

What is the Median?
The median is the middle value of a sorted array.



Naïve Approach:

1. Sort the array.
2. Find the element in the middle position.

Time complexity:

- Sorting: O(n log n)
- Selecting the middle element: O(1)
- Total: O(n log n)

PES UNIVERSITY

Computing a Median and the Selection Problem

Issues with the Naïve Approach:

- Sorting places all elements in their correct positions.
- We only need the middle element, so sorting the entire array is unnecessary.

What can be optimized:

- Instead of sorting all elements, directly try to find the element at the n/2 position.
- This problem of finding the k-th smallest or largest element is called the **Selection Problem.**

Proposed Solution:

- QuickSelect to solve the Selection Problem

PES UNIVERSITY

Computing a Median and the Selection Problem

Quick Select:

- Similar to Quick Sort, we pick a **pivot** element and **partition** the array:
  - Elements **smaller** than the pivot go to the **left**.
  - Elements **larger** than the pivot go to the **right**.
  - The **pivot** is placed in its **correct** position.
  - This partition technique is called **Lomuto Partition**.

- Difference from Quick Sort:
  - Quick Sort recursively sorts the entire array.
  - Quick Select **stops** once the pivot is at the k-th position (n/2 for median).

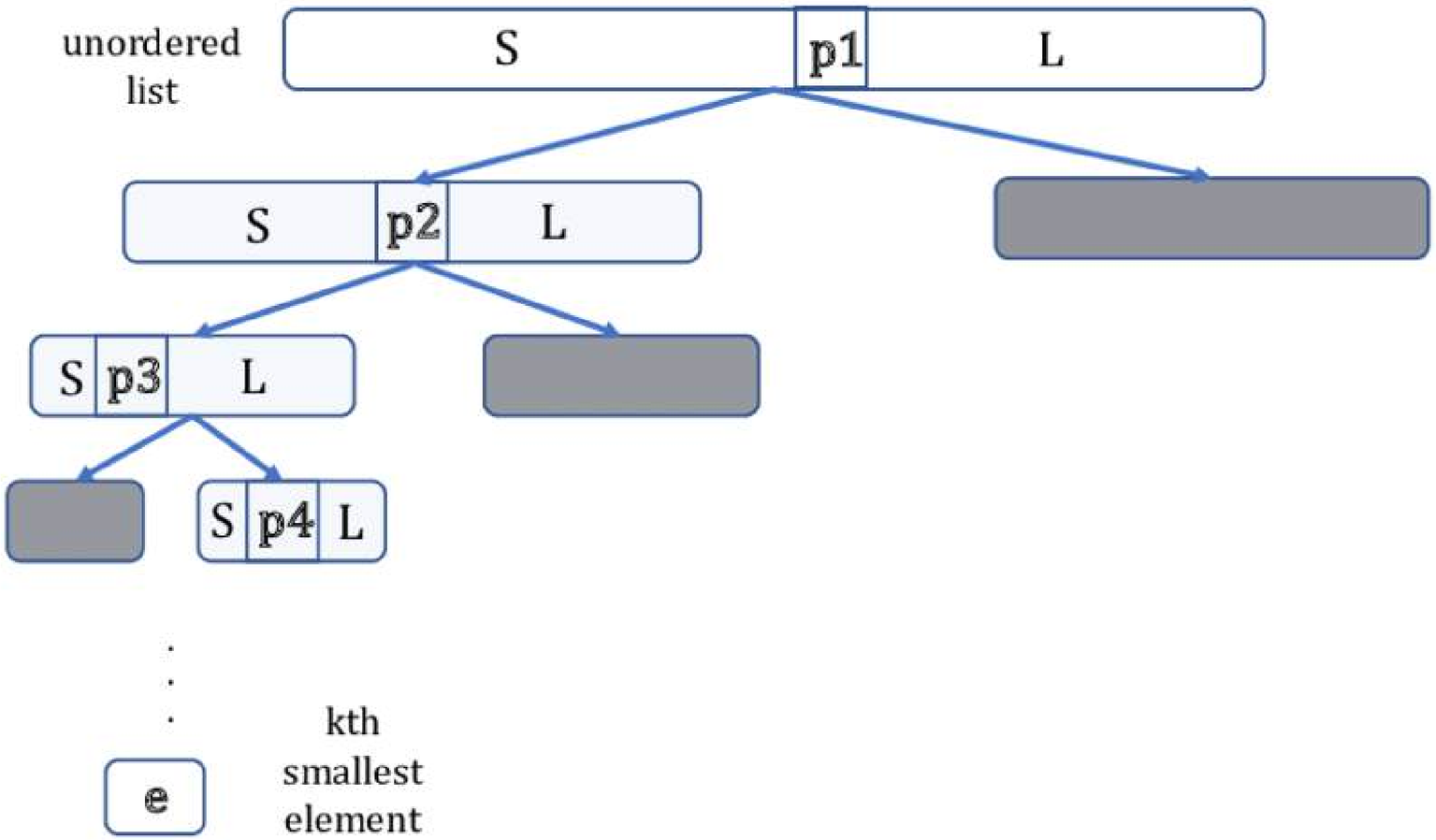
Algorithm for QuickSelect

Input : List, left is first position of list, right is last position of list and k is k-th smallest element.
Output : A new list is partitioned.

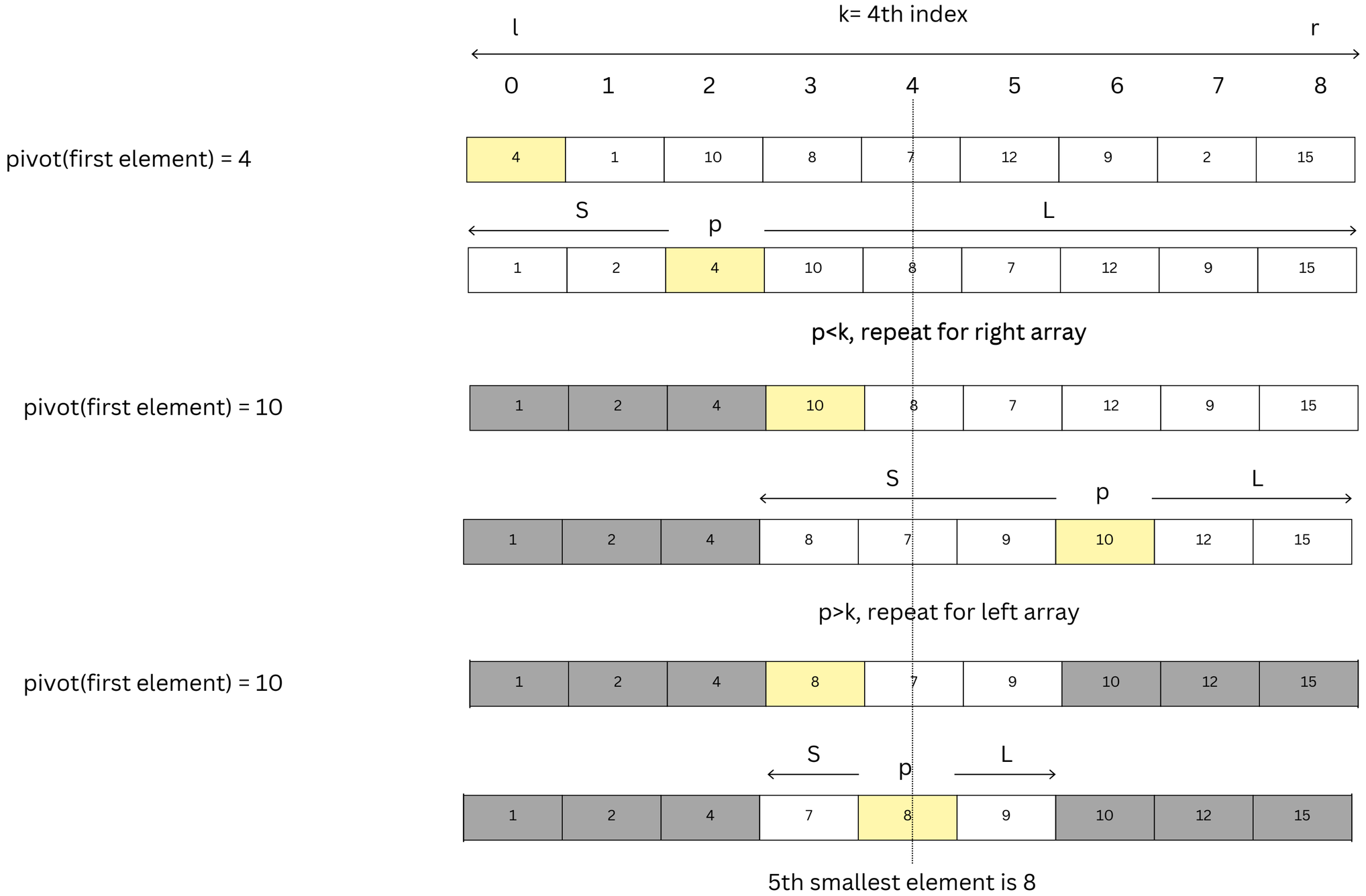
```
quickSelect(list, left, right, k)
   if left = right
      return list[left]

   // Select a pivotIndex between left and right
   pivotIndex <= partition(list, left, right, pivotIndex)
   if k = pivotIndex
      return list[k]
   else if k < pivotIndex
      right <= pivotIndex - 1
   else
      left <= pivotIndex + 1
```

QuickSelect



Find the 5th smallest element in the given array



Time Complexities for QuickSelect

Best Case:

- The best case occurs when the pivot chosen is always the median of the array. This perfectly divides the array in half at each step.
- The recurrence relation: $T(n)=T(n/2)+O(n)$
- Solving this gives $O(n)$ time complexity, and the problem size is reduced exponentially.

Worst Case:

- The worst case occurs when the pivot selection is poor, e.g., always choosing the smallest or largest element. This results in highly unbalanced partitions, where one side has n - 1 elements and the other has 0.
- The recurrence becomes: $T(n)=T(n-1)+O(n)$
- Solving this gives $O(n^2)$ time complexity.

Improve the Worst Case Complexity

There are two key ways to achieve better worst-case complexity:

1. Choosing a Better Pivot Selection Strategy such as Median of Medians Algorithm.
2. Using an Improved Partitioning Algorithm such as using Hoare's Partition Scheme instead of Lomuto's Partition Scheme can improve efficiency by reducing swaps.

Median of Medians Algorithm (Guaranteed O(n) Worst-Case):

We need to **choose a pivot close to the median** consistently. This ensures better partitioning and prevents highly unbalanced cases.

- Divide the array into groups of 5 elements (or another small fixed size).
- Find the median of each group.
- Recursively find the median of these medians to use as the pivot.
- Ensures a pivot that is reasonably close to the true median, avoiding worst-case $O(n^2)$.
- Guarantees O(n) worst-case time complexity.

Variable Decrease and Conquer

By utilizing **variable-size Decrease and Conquer**, we optimized the Selection Algorithm, making median computation more efficient.

Unlike traditional divide-and-conquer, where the problem size is reduced by a fixed factor (e.g., halving in binary search), **variable-size** reduction allows for **adaptive problem** shrinking at each step. In Quick Select, this means:

- Instead of processing the entire array, we eliminate a portion based on the pivot's position.
- The size reduction varies **dynamically**, depending on where the pivot is placed.
- This flexibility enables an **O(n)** average-time complexity, significantly improving over naive sorting methods.

By using variable Decrease and Conquer, we were able to directly target the element we needed, making the solution far more efficient.

1.3 Master theorem

The master theorem is a formula for solving recurrences of the form $T(n) = aT(n/b) + f(n)$, where $a \geq 1$ and $b > 1$ and $f(n)$ is asymptotically positive. (Asymptotically positive means that the function is positive for all sufficiently large n.)

This recurrence describes an algorithm that divides a problem of size n into a subproblems, each of size n/b, and solves them recursively. (Note that n/b might not be an integer, but in section 4.6 of the book, they prove that replacing $T(n/b)$ with $T(\lfloor n/b \rfloor)$ or $T(\lceil n/b \rceil)$ does not affect the asymptotic behavior of the recurrence. So we will just ignore floors and ceilings here.)

The theorem is as follows:

> ***Theorem 4.1 (Master theorem)***
> Let $a \geq 1$ and $b > 1$ be constants, let $f(n)$ be a function, and let $T(n)$ be defined on the nonnegative integers by the recurrence
>
> $$T(n) = aT(n/b) + f(n)\,,$$
>
> where we interpret n/b to mean either $\lfloor n/b \rfloor$ or $\lceil n/b \rceil$. Then $T(n)$ has the following asymptotic bounds:
>
> 1. If $f(n) = O(n^{\log_b a - \epsilon})$ for some constant $\epsilon > 0$, then $T(n) = \Theta(n^{\log_b a})$.
> 2. If $f(n) = \Theta(n^{\log_b a})$, then $T(n) = \Theta(n^{\log_b a} \lg n)$.
> 3. If $f(n) = \Omega(n^{\log_b a + \epsilon})$ for some constant $\epsilon > 0$, and if $af(n/b) \leq cf(n)$ for some constant $c < 1$ and all sufficiently large n, then $T(n) = \Theta(f(n))$. ∎

The master theorem compares the function $n^{\log_b a}$ to the function $f(n)$. Intuitively, if $n^{\log_b a}$ is larger (by a polynomial factor), then the solution is $T(n) = \Theta(n^{\log_b a})$. If $f(n)$ is larger (by a polynomial factor), then the solution is $T(n) = \Theta(f(n))$. If they are the same size, then we multiply by a logarithmic factor.

Be warned that these cases are not exhaustive – for example, it is possible for $f(n)$ to be asymptotically larger than $n^{\log_b a}$, but not larger by a polynomial factor (no matter how small the exponent in the polynomial is). For example, this is true when $f(n) = n^{\log_b a} \log n$. In this situation, the master theorem would not apply, and you would have to use another method to solve the recurrence.

1.3.1 Examples:

To use the master theorem, we simply plug the numbers into the formula.

Example 1: $T(n) = 9T(n/3)+n$. Here $a = 9$, $b = 3$, $f(n) = n$, and $n^{\log_b a} = n^{\log_3 9} = \Theta(n^2)$. Since $f(n) = O(n^{\log_3 9-\epsilon})$ for $\epsilon = 1$, case 1 of the master theorem applies, and the solution is $T(n) = \Theta(n^2)$.

Example 2: $T(n) = T(2n/3)+1$. Here $a = 1$, $b = 3/2$, $f(n) = 1$, and $n^{\log_b a} = n^0 = 1$. Since $f(n) = \Theta(n^{\log_b a})$, case 2 of the master theorem applies, so the solution is $T(n) = \Theta(\log n)$.

Example 3: $T(n) = 3T(n/4) + n\log n$. Here $n^{\log_b a} = n^{\log_4 3} = O(n^{0.793})$. For $\epsilon = 0.2$, we have $f(n) = \Omega(n^{\log_4 3+\epsilon})$. So case 3 applies if we can show that $af(n/b) \leq cf(n)$ for some $c < 1$ and all sufficiently large n. This would mean $3\frac{n}{4}\log\frac{n}{4} \leq cn\log n$. Setting $c = 3/4$ would cause this condition to be satisfied.

Example 4: $T(n) = 2T(n/2)+n\log n$. Here the master method does not apply. $n^{\log_b a} = n$, and $f(n) = n\log n$. Case 3 does not apply because even though $n\log n$ is asymptotically larger than n, it is not polynomially larger. That is, the ratio $f(n)/n^{\log_b a} = \log n$ is asymptotically less than n^ϵ for all positive constants ϵ.

Multiplication of large integers - Idea

- Let the two numbers being multiplied be a and b
- a and b are n–digit integers, where n is a positive even number
- Let the first half of a's digits be a_1 and second half be a_0
- Similarly, let the first half of b's digits be b_1 and second half be b_0
- In these notations, $a = a_1a_0$ implies $a = a_1*10^{n/2} + a_0$ and $b = b_1b_0$ implies $b = b_1*10^{n/2} + b^0$

Multiplication of large integers - Idea

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

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

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

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

Multiplication of large integers - Analysis

- $M(n) = 3M(n/2)$ for $n > 1$, $M(1) = 1$
- Solving it by backward substitutions for $n = 2^k$ yields:

$M(2^k) = 3M(2^{k-1}) = 3[3M(2^{k-2})] = 3^2M(2^{k-2})$

$= \ldots = 3^i M(2^{k-i}) = \ldots = 3^kM(2^{k-k}) = 3^k$

- Since $k = \log_2 n$: $M(n) = 3^{\log_2 n} = n^{\log_2 3} = n^{1.585}$
- The number of additions is given by:

$A(n) = 3A(n/2) + cn$ for $n > 1$, $A(1) = 1$

$A(n)$ belongs to $\Theta(n^{\log_2 3})$

Multiplication of large integers - Example

2135 * 4014

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

$= (21*40)*10^4 + c1*10^2 + 35*14$

where c1 = (21+35)*(40+14) - 21*40 - 35*14, and

21*40 = (2*10 + 1) * (4*10 + 0)

$= (2*4)*10^2 + c2*10 + 1*0$

where c2 = (2+1)*(4+0) - 2*4 - 1*0, etc.,

Strassen's Matrix Multiplication

- This algorithm was published by V Strassen in 1969
- The principal insight of the algorithm lies in the discovery that we can find product of two 2 – by – 2 matrices A and B with seven multiplications as opposed to the eight required by the Brute – Force algorithm
- This is accomplished by the following formulae:

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

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

Strassen's Matrix Multiplication

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

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

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

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

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

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

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

Strassen's Matrix Multiplication – General Formula

For any two matrices A and B of size n – by – n, we can divide A, B and the product C as follows:

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

The sub – matrices can be treated as numbers to get the correct product

Strassen's Matrix Multiplication – Analysis

- If M(n) is the number of multiplications made by Strassen's algorithm in multiplying two matrices n – by – n, we get the following recurrence relation for it:

$M(n) = 7M(n/2)$ for $n>1$, $M(1) = 1$

Since $n = 2^k$,

$$M(2^k) = 7M(2^{k-1}) = 7[7M(2^{k-2})] = 7^2M(2^{k-2}) = \ldots$$
$$= 7^i\, M(2^{k-i}) \ldots = 7^k\, M\,(2^{k-k}) = 7^k$$

Since $k = \log_2 n$,

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

Strassen's Matrix Multiplication – Analysis

- The number of additions are given by the following recurrence:

  $A(n) = 7\,A(n/2) + 18(n/2)^2$ for $n>1$, $A(1) = 0$

- According to Master's Theorem, A(n) belongs to $\Theta(n^{\log_2 7})$

Decrease and Conquer

- **Decrease** or reduce problem instance to smaller instance of the same problem and extend solution.
- **Conquer** the problem by solving smaller instance of the problem.
- **Extend** solution of smaller instance to obtain solution to original problem .
- **Exploit** the relationship between a solution to a given instance of a problem and a solution to its smaller instance.

- Can be implemented either top-down or bottom-up
- Also referred to as *inductive* or *incremental* approach

Decrease and Conquer

3 Types of Decrease and Conquer

Decrease by a constant (usually by 1):
- insertion sort
- graph traversal algorithms (DFS and BFS)
- 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

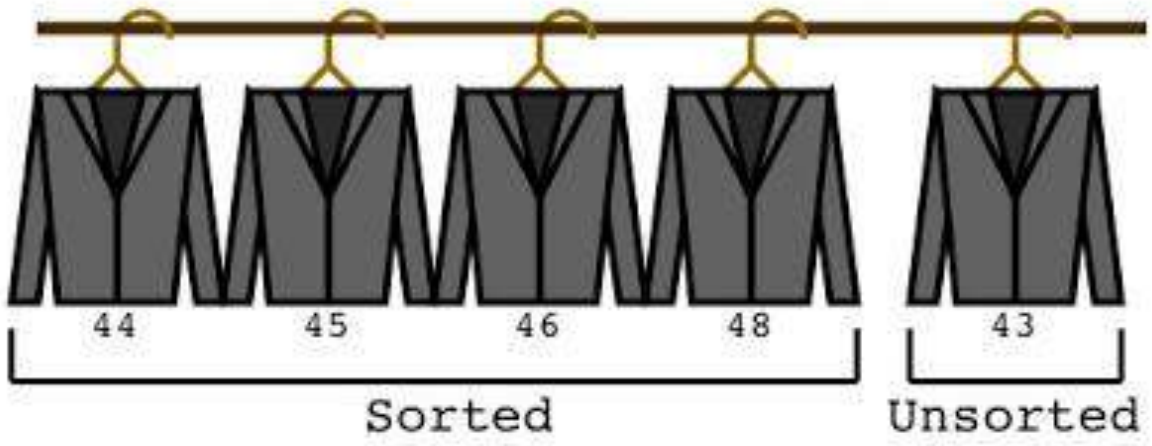
Variable-size decrease
- Euclid's algorithm
- selection by partition
- Nim-like games

This usually results in a recursive algorithm.

Insertion Sort

Imagine a card game

Cards in your hand are sorted.
The dealer hands you exactly one new card.
How would you rearrange your cards



Insertion Sort

- Insertion sort is based on the idea that one element from the input elements is consumed in each iteration to find its correct position i.e, the position to which it belongs in a sorted array.
- grows the sorted array at each iteration
- compares the current element with the largest value in the sorted array.
- If the current element is greater, then it leaves the element in its place and moves on to the next element else it finds its correct position in the sorted array and moves it to that position.
- This is done by shifting all the elements, which are larger than the current element, in the sorted array to one position ahead

Insertion Sort

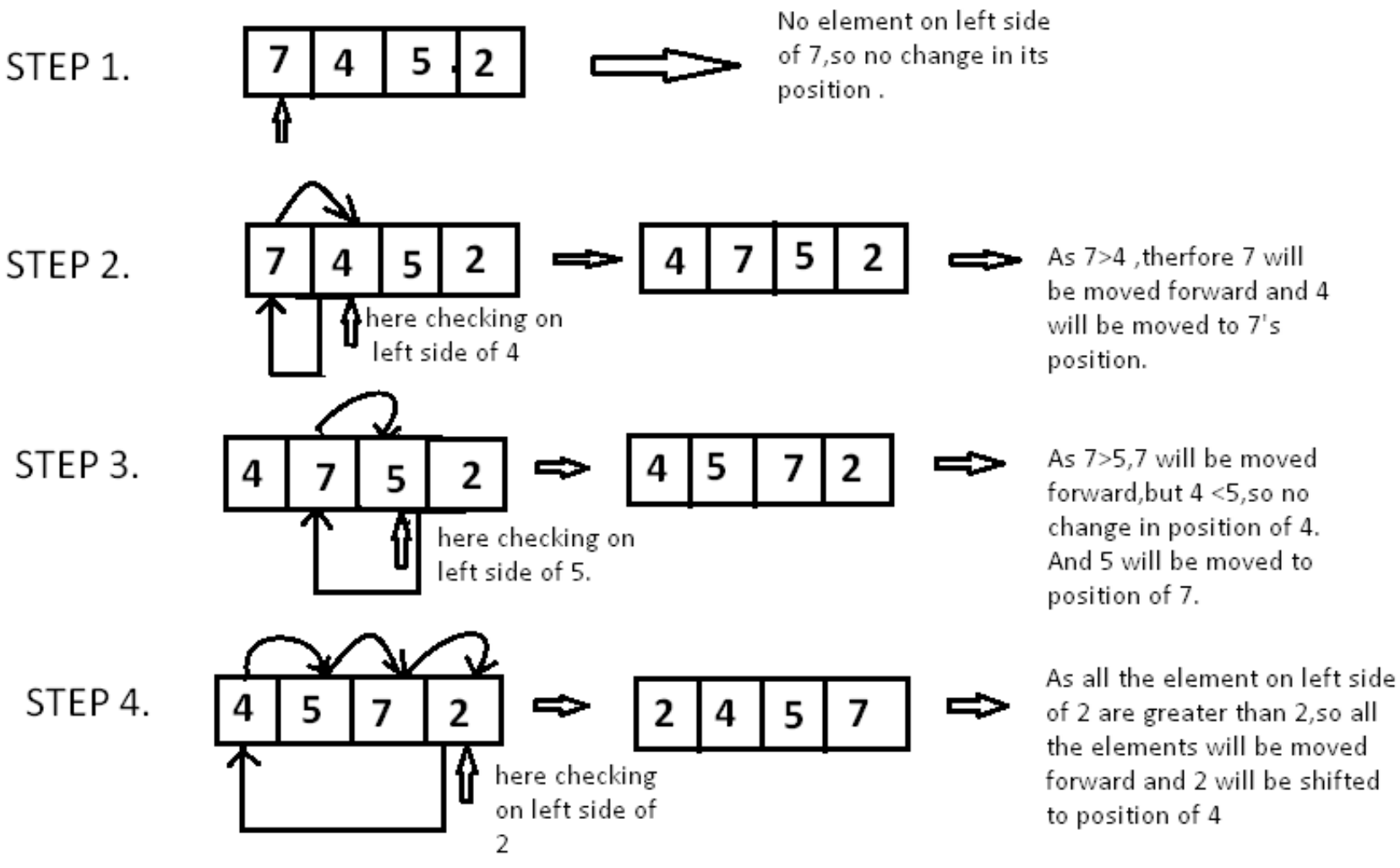
To sort array A[0..*n*-1], sort A[0..*n*-2] recursively and then insert A[*n*-1] in its proper place among the sorted A[0..*n*-2]

Usually implemented bottom up (nonrecursively)

Example: Sort 6, 4, 1, 8, 5

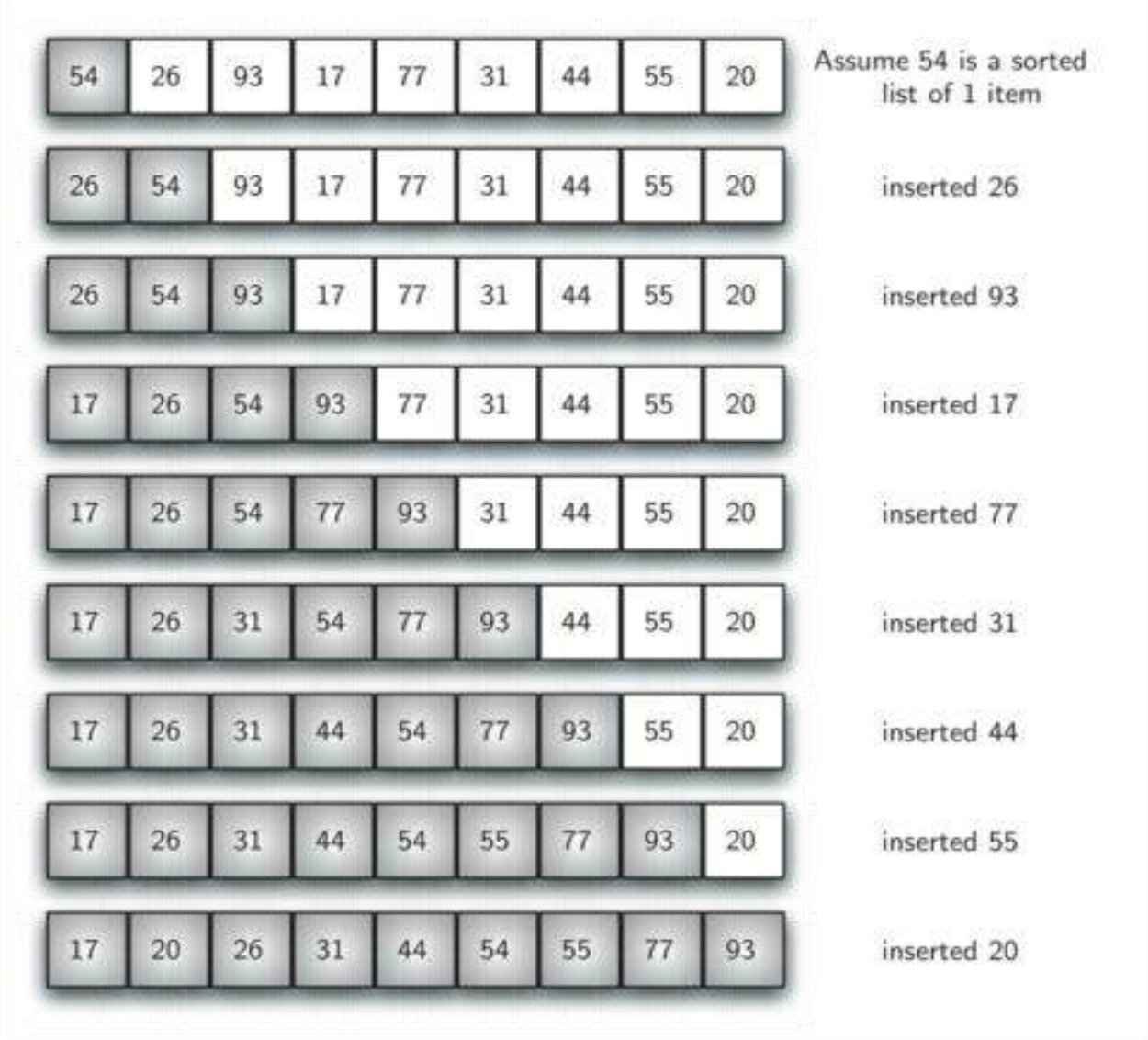
6 | 4 1 8 5
4 6 | 1 8 5
1 4 6 | 8 5
1 4 6 8 | 5
1 4 5 6 8

Insertion Sort



PES UNIVERSITY

Insertion Sort



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

```
//Sorts a given array by insertion sort
//Input: An array A[0..n − 1] of n orderable elements
//Output: Array A[0..n − 1] sorted in nondecreasing order
for i ← 1 to n − 1 do
    v ← A[i]
    j ← i − 1
    while j ≥ 0 and A[j] > v do
        A[j + 1] ← A[j]
        j ← j − 1
    A[j + 1] ← v
```

Time efficiency

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

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

$C_{best}(n) = n - 1 \in \Theta(n)$ (also fast on almost sorted arrays)

Space efficiency: in-place

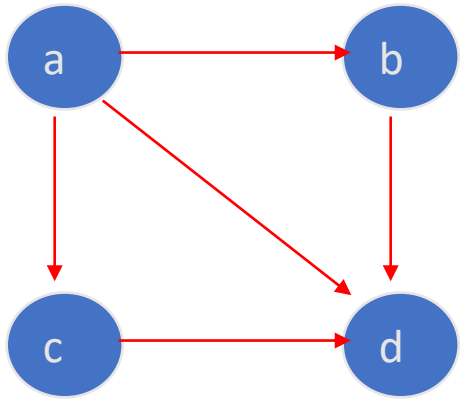
Stability: yes

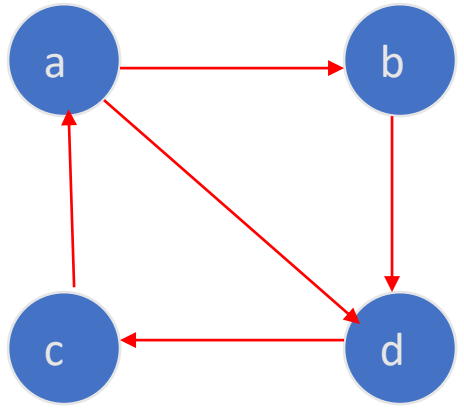
Best elementary sorting algorithm overall

Binary insertion sort

DAGs and Topological Sorting

DAG: a directed acyclic graph, i.e. a directed graph with no (directed) cycles





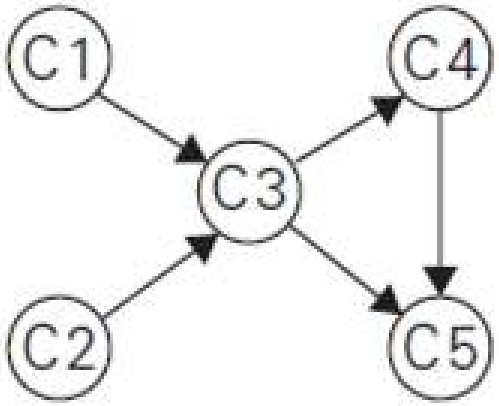
Arise in modeling many problems that involve prerequisite constraints (construction projects, document version control)

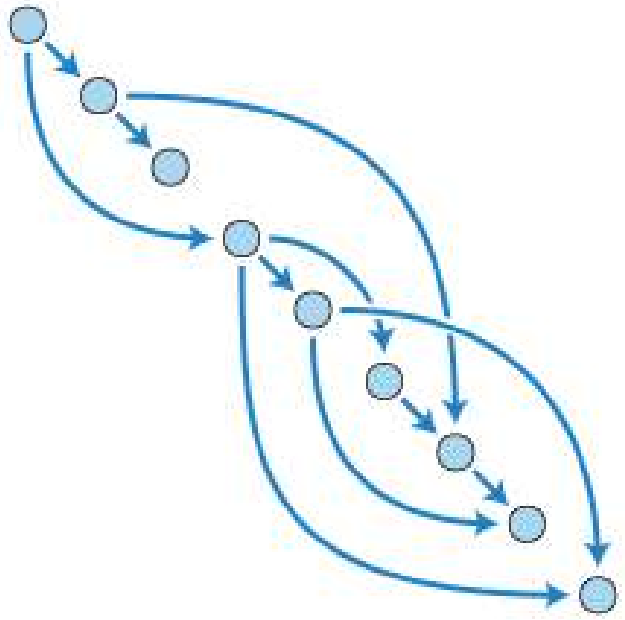
Vertices of a dag can be linearly ordered so that for every edge its starting vertex is listed before its ending vertex (*topological sorting*). Being a dag is also a necessary condition for topological sorting to be possible.

Topological Sorting

Topological Sorting: is listing vertices of a directed graph 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.

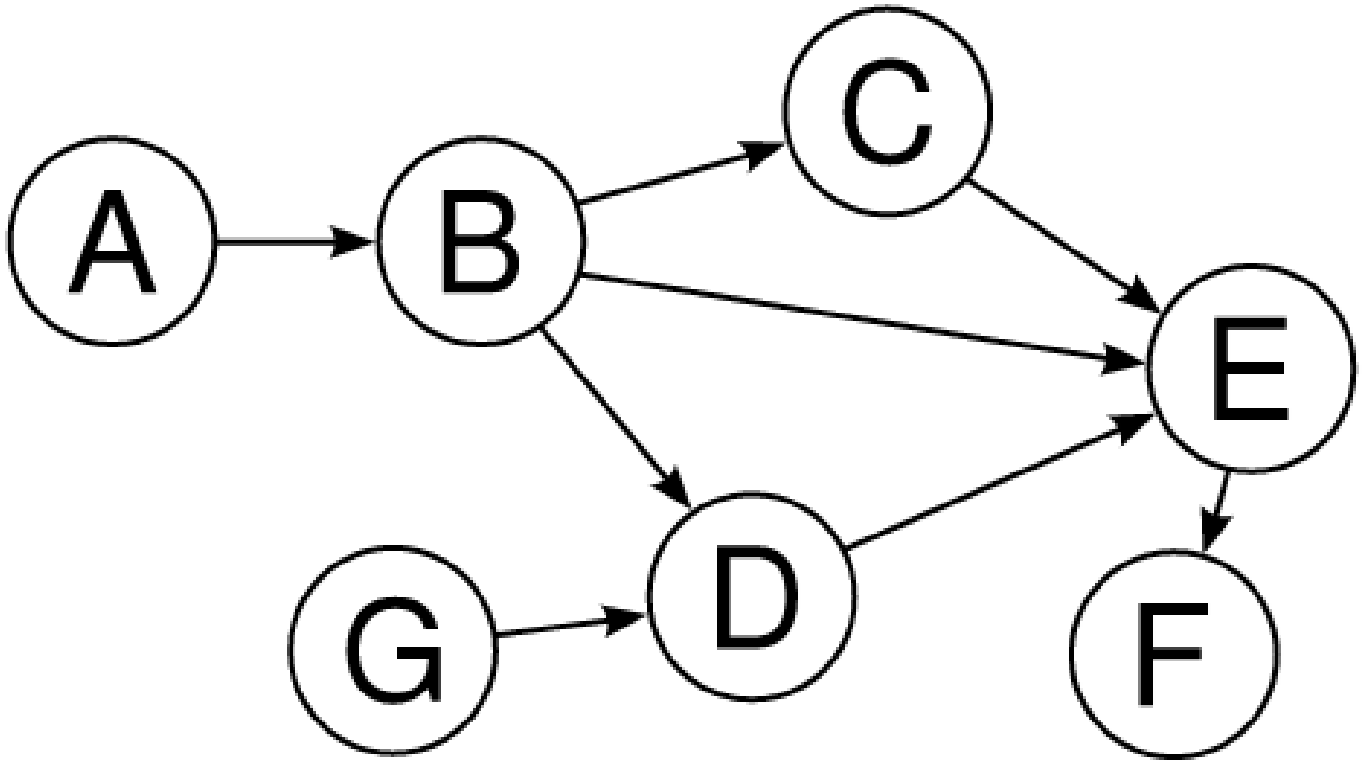
A digraph has a topological sorting iff it is a **dag**.





Finding a **Topological Sorting** of the vertices of a dag:

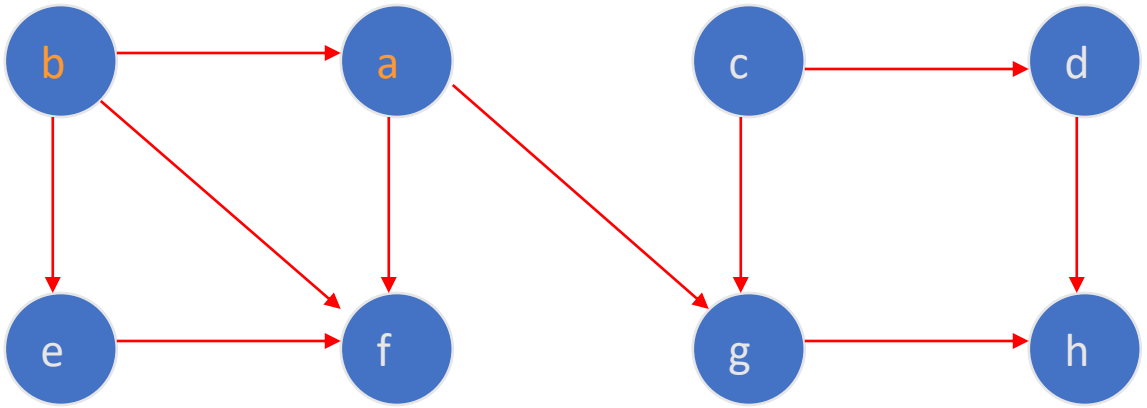
- **DFS-based** algorithm
- **Source-removal** algorithm



DFS-based algorithm for topological sorting

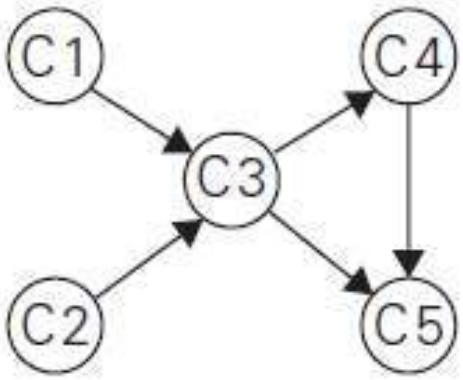
- Perform DFS traversal, noting the order vertices are popped off the traversal stack
- Reverse order solves topological sorting problem
- Back edges encountered?→ NOT a dag!

Example:



Efficiency: The same as that of DFS.

DFS-based algorithm for finding **Topological Sorting**

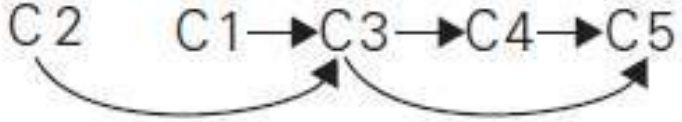


(a)

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

(b)

The popping-off order:
C5, C4, C3, C1, C2
The topologically sorted list:

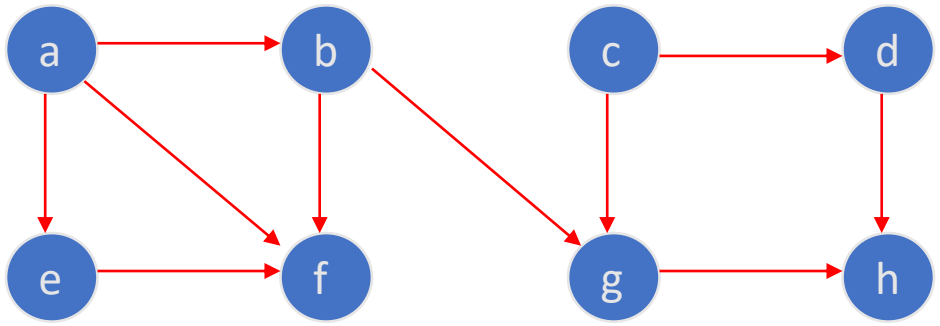


(c)

Source removal algorithm

Repeatedly identify and remove a *source* (a vertex with no incoming edges) and all the edges incident to it until either no vertex is left or there is no source among the remaining vertices (not a dag)

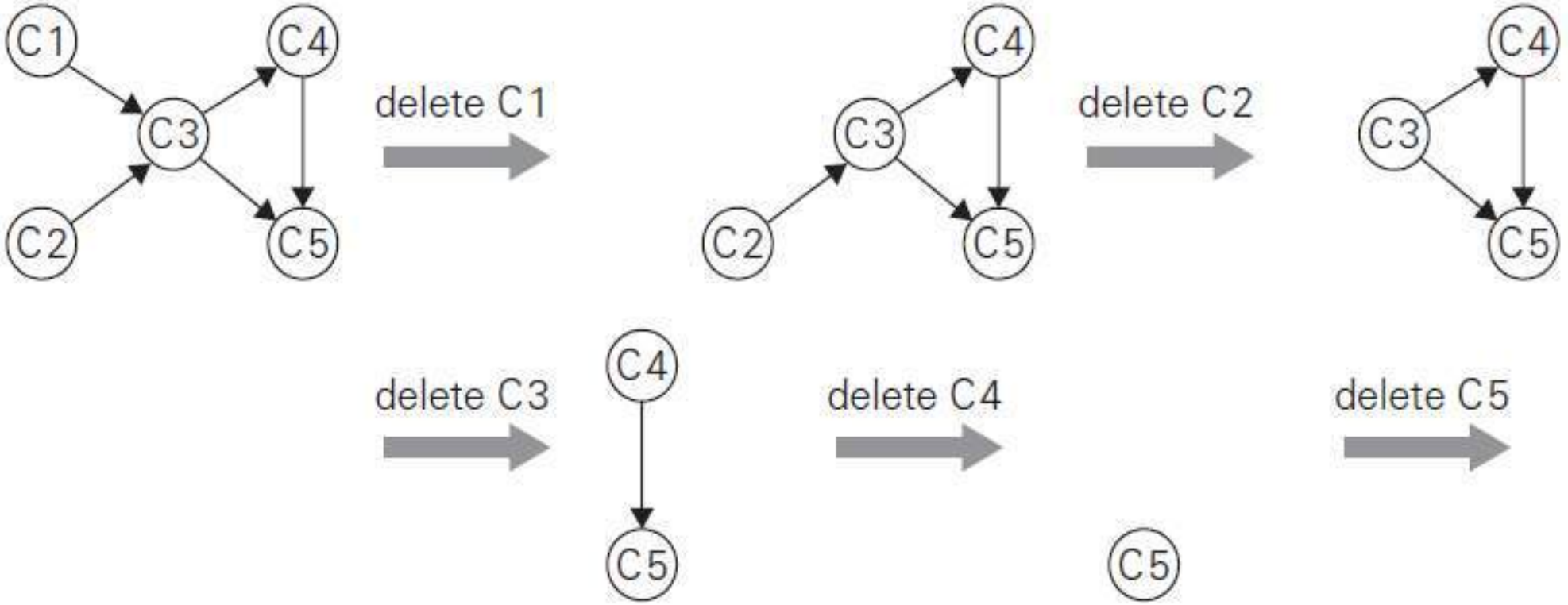
Example:



"Invert" the adjacency lists for each vertex to count the number of incoming edges by going thru each adjacency list and counting the number of times that each vertex appears in these lists. To remove a source, decrement the count of each of its neighbors by one.

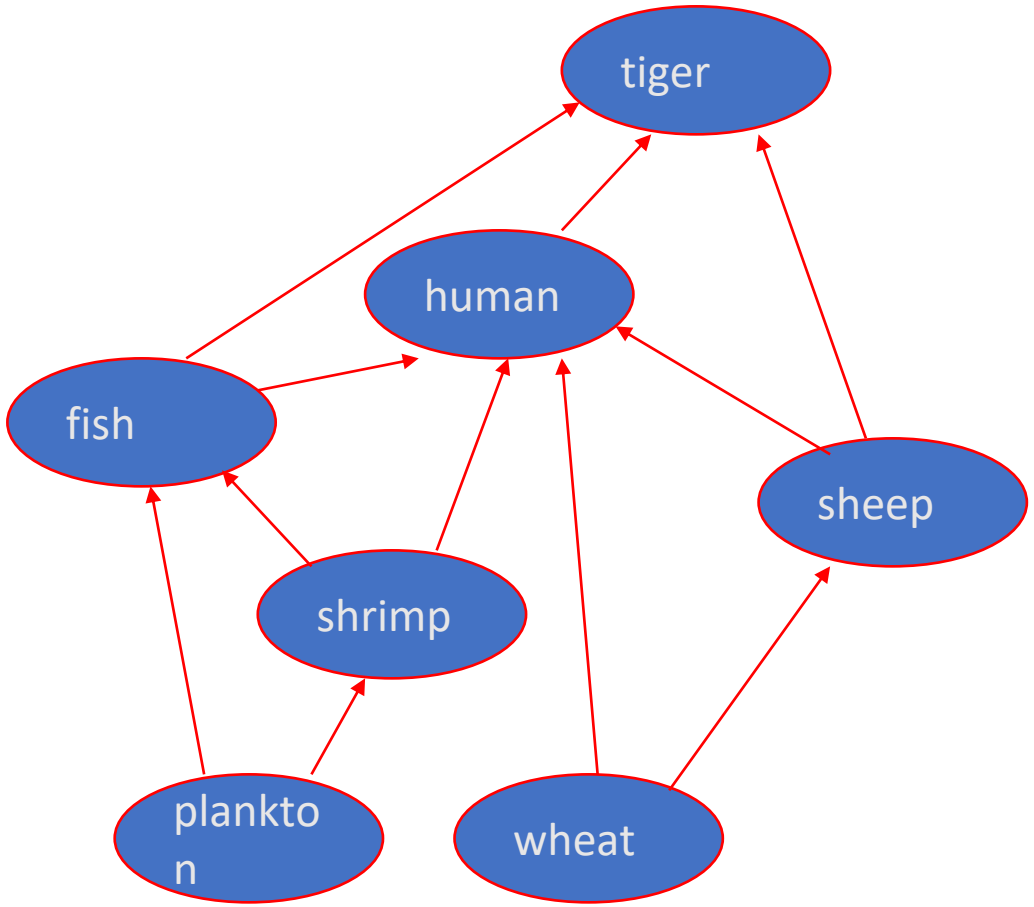
```
Algorithm SourceRemoval_Toposort(V, E)
L ← Empty list that will contain the sorted vertices
S ← Set of all vertices with no incoming edges
while S is non-empty do
    remove a vertex v from S
    add v to tail of L
    for each vertex m with an edge e from v to m do
        remove edge e from the graph
        if m has no other incoming edges then
            insert m into S
if graph has edges then
    return error (not a DAG)
else return L (a topologically sorted order)
```

Decrease and Conquer



The solution obtained is C1, C2, C3, C4, C5

Order the following items in a food chain



Combinatorial Objects

- Permutations
- Combinations
- Subsets of a given set

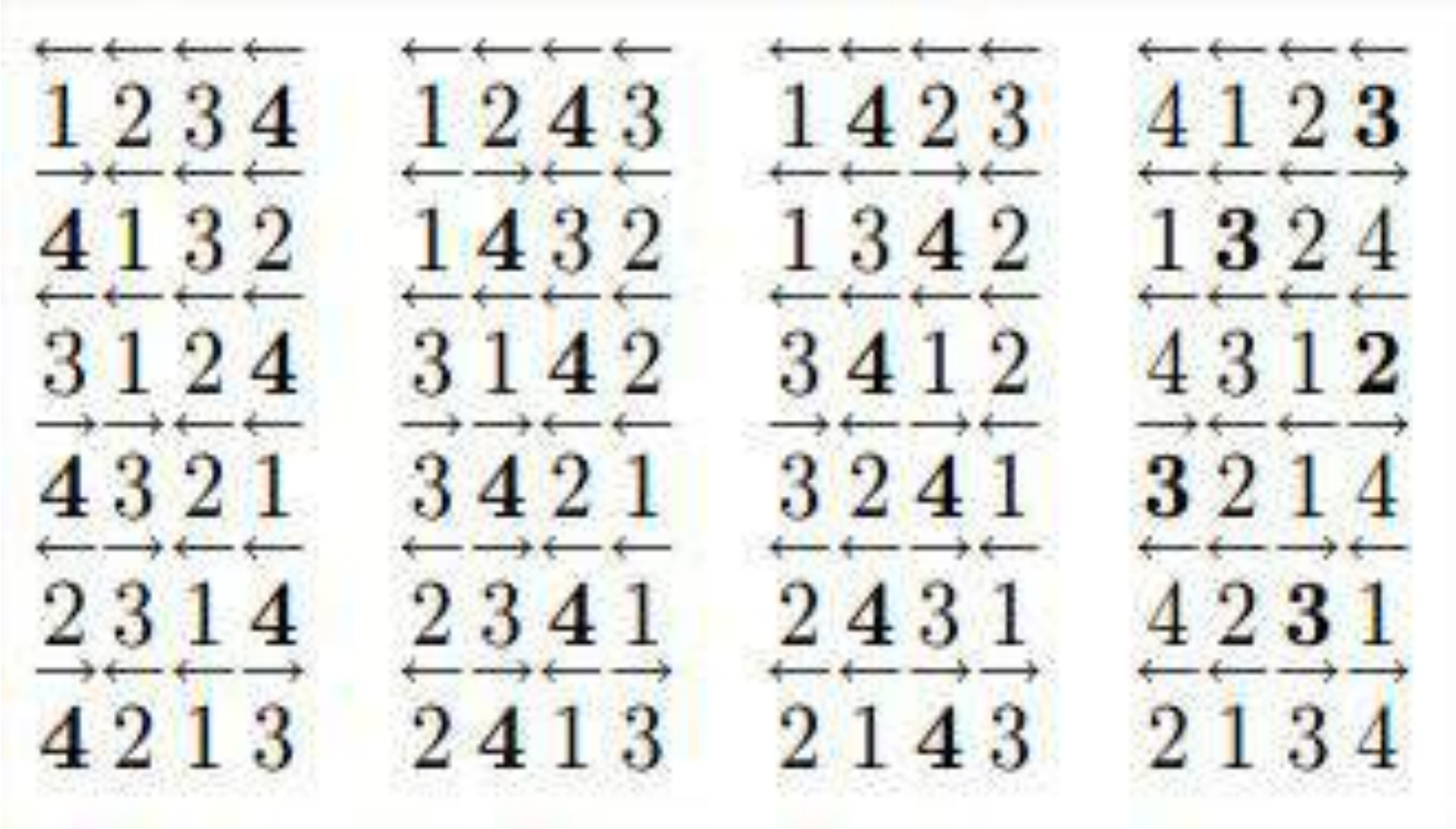
Generating Permutations

- Underlying set elements are to be permuted
- Decrease and conquer approach
- Satisfies the minimal change requirement
- Example: Johnson- Trotter algorithm

Generating Permutations

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


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

Permutations in Lexicographic order

ALGORITHM *LexicographicPermute*(n)

//Generates permutations in lexicographic order
//Input: A positive integer n
//Output: A list of all permutations of $\{1, \ldots, n\}$ in lexicographic order
initialize the first permutation with $12 \ldots n$
while last permutation has two consecutive elements in increasing order **do**
    let i be its largest index such that $a_i < a_{i+1}$ $\quad //a_{i+1} > a_{i+2} > \cdots > a_n$
    find the largest index j such that $a_i < a_j$ $\quad //j \geq i+1$ since $a_i < a_{i+1}$
    swap a_i with a_j $\quad //a_{i+1}a_{i+2} \ldots a_n$ will remain in decreasing order
    reverse the order of the elements from a_{i+1} to a_n inclusive
    add the new permutation to the list

Generating Subsets:

Knapsack problem needed to find the most valuable subset of items that fits a knapsack of a given capacity.

Powerset: set of all subsets of a set. Set A={1, 2, ..., n} has $\mathbf{2^n}$ subsets.

Generate all subsets of the set A={1, 2, ..., n}.

Any **decrease-by-one** idea?
# of subsets of { } = 2^0 = 1, which is **{ }** itself
Suppose, we know how to generate all subsets of {1,2,...,n-1}
Now, how can we generate all subsets of {1,2,...,n} ?

Generating Subsets:

All subsets of {1,2,...,n-1}: 2^{n-1} such subsets

All subsets of {1,2,...,n}:
2^{n-1} subsets of {1,2,...,n-1} and
another 2^{n-1} subsets of {1,2,...,n-1} having **'n'** with them.

That adds up to all 2^n subsets of {1,2,...,n}

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

Alternate way of Generating Subsets:

Knowing the binary nature of either having **n**th element or not, any idea involving binary numbers itself?

One-to-one correspondence between all 2^n bit strings $b_1b_2...b_n$
and 2^n subsets of $\{a_1, a_2, \ldots, a_n\}$.

Each bit string $b_1b_2...b_n$ could correspond to a subset.
In a bit string $b_1b_2...b_n$, depending on whether b_i is 1 or 0, a_i is in the subset or not in the subset.

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

Generating Subsets in Squashed order:

Squashed order: any subset involving a_j can be listed only after all the subsets involving $a_1, a_2, \ldots, a_{j-1}$

Both of the previous methods does generate subsets in squashed order.

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

Generating Subsets in **Squashed order:**

Squashed order: any subset involving a_j can be listed only after all the subsets involving $a_1, a_2, \ldots, a_{j-1}$

Can we do it with minimal change in bit-string (actually, just one-bit change to get the next bit string)? This would mean, to get a new subset, just change one item (remove one item or add one item).

Binary reflected gray code:

000 001 011 010 110 111 101 100

Merge Sort - Idea

- Split array A[0..n-1] into 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

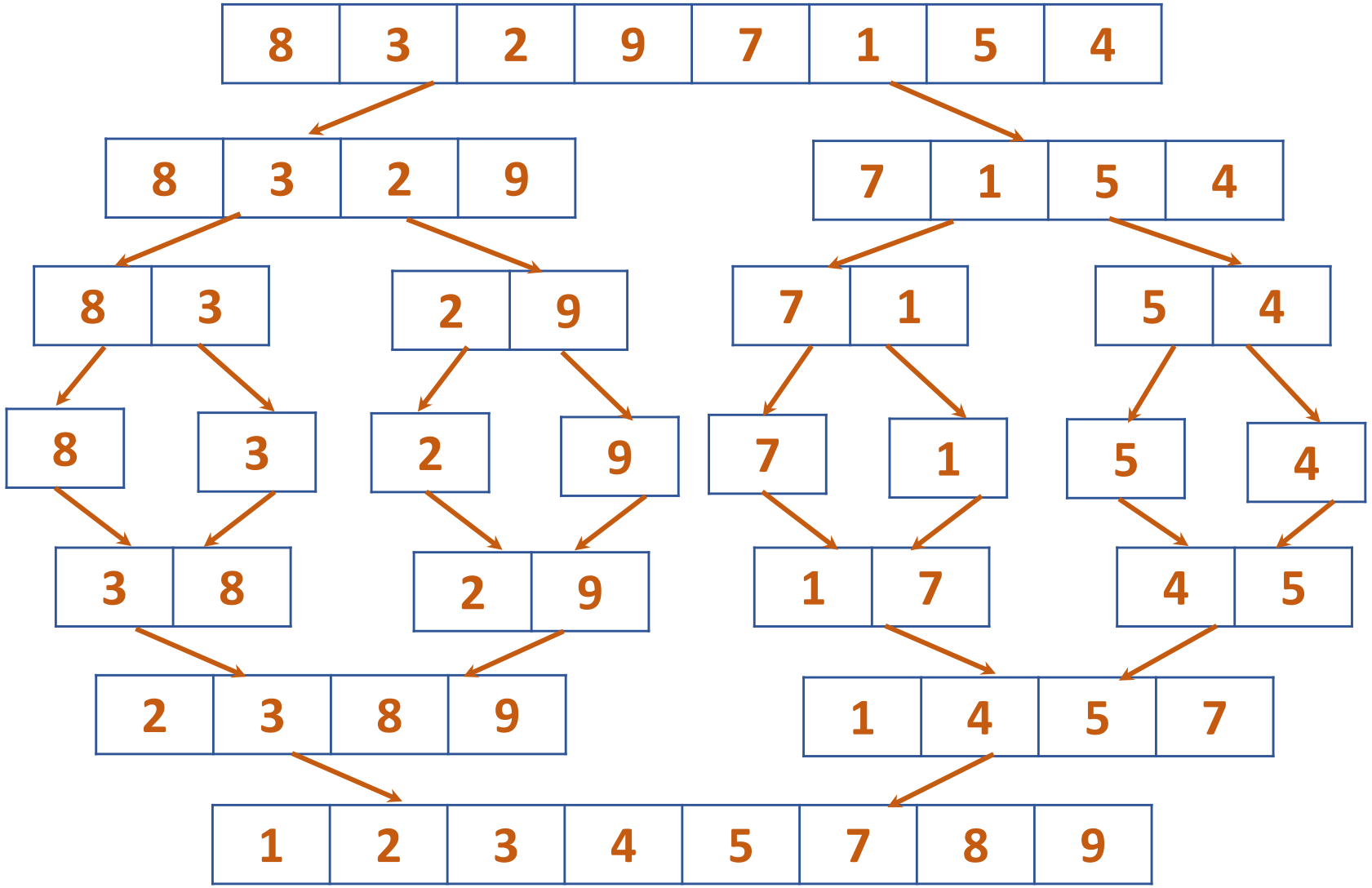
Merge Sort - Algorithm

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 non decreasing 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)

Merge Sort - Algorithm

```
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<-0;  j<-0;  k<-0
while i < p and j < q do
   if B[i] ≤ C[j]
            A[k] <-B[i];  i <-i  + 1
   else   A[k]<-C[j];   j<-j+1
   k<-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]
```

Merge Sort - Example



Merge Sort - Algorithm

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

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

- The number of key comparisons performed during the merging stage in the worst case is:

$$C_{merge}(n) = n - 1$$

- Using the above equation:

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

- Applying Master Theorem to the above equation:

$$C_{worst}(n) \in \Theta(n \log n)$$

DESIGN AND ANALYSIS OF ALGORITHMS

Binary Tree

- A *binary tree* **T** is defined as a finite set of nodes that is either empty or consists of a root and two disjoint binary trees $\mathbf{T_L}$ and $\mathbf{T_R}$ called as the left and right subtree of the root

- The definition itself divides the Binary Tree into two smaller structures and hence many problems concerning the binary trees can be solved using the Divide – And – Conquer technique

- The binary tree is a Divide – And – Conquer ready structure ☺

PES UNIVERSITY

Height of a Binary Tree

- Height of a Binary Tree: Length of the longest path from root to leaf

ALGORITHM Height(T)

//Computes recursively the height of a binary tree

//Input: A binary tree T

//Output: The height of T

if T = Ø return -1

else return max(Height(T_L), Height(T_R))+ 1

DESIGN AND ANALYSIS OF ALGORITHMS

Height of a Binary Tree - Analysis

- The measure of input's size is the number of nodes in the given binary tree. Let us represent this number as n(T)
- Basic Operation: Addition
- The recurrence relation is setup as follows:

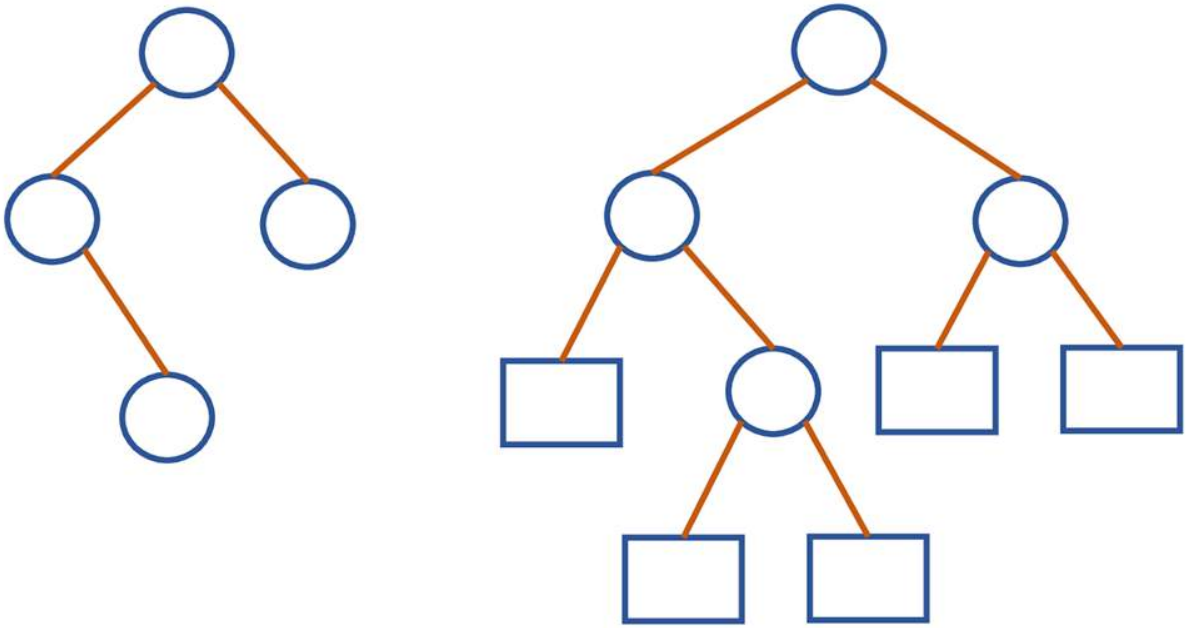
$$A(n(T)) = A(n(T_L)) + A(n(T_R)) + 1, \quad \text{for } n(T) > 0$$

$$A(0) = 0$$

DESIGN AND ANALYSIS OF ALGORITHMS

Height of a Binary Tree - Analysis

- In the analysis of tree algorithms, the tree is extended by replacing empty subtrees by special nodes called external nodes



DESIGN AND ANALYSIS OF ALGORITHMS

Height of a Binary Tree - Analysis

- x – Number of external nodes
- n – Number of internal nodes

$$x = n + 1$$

- The number of comparisons to check whether a tree is empty or not:

$$C(n) = n + x = 2n + 1$$

- The number of additions is:

$$A(n) = n$$

DESIGN AND ANALYSIS OF ALGORITHMS

Binary Tree Traversals

- The three classic traversals for a binary tree are inorder, preorder and postorder traversals

- In the preorder traversal, the root is visited before the left and right subtrees are visited (in that order)
- In the inorder traversal, the root is visited after visiting its left subtree but before visiting the right subtree
- In the postorder traversal, the root is visited after visiting the left and right subtrees (in that order)

DESIGN AND ANALYSIS OF ALGORITHMS

Binary Tree Traversals

Algorithm Inorder(T)
if $T \neq \varnothing$
    Inorder(Tleft)
    print(root of T)
    Inorder(Tright)

Algorithm Preorder(T)
if $T \neq \varnothing$
    print(root of T)
    Preorder(T_{left})
    Preorder(T_{right})

Algorithm Postorder(T)
if $T \neq \varnothing$
    Postorder(T_{left})
    Postorder(T_{right})
    print(root of T)