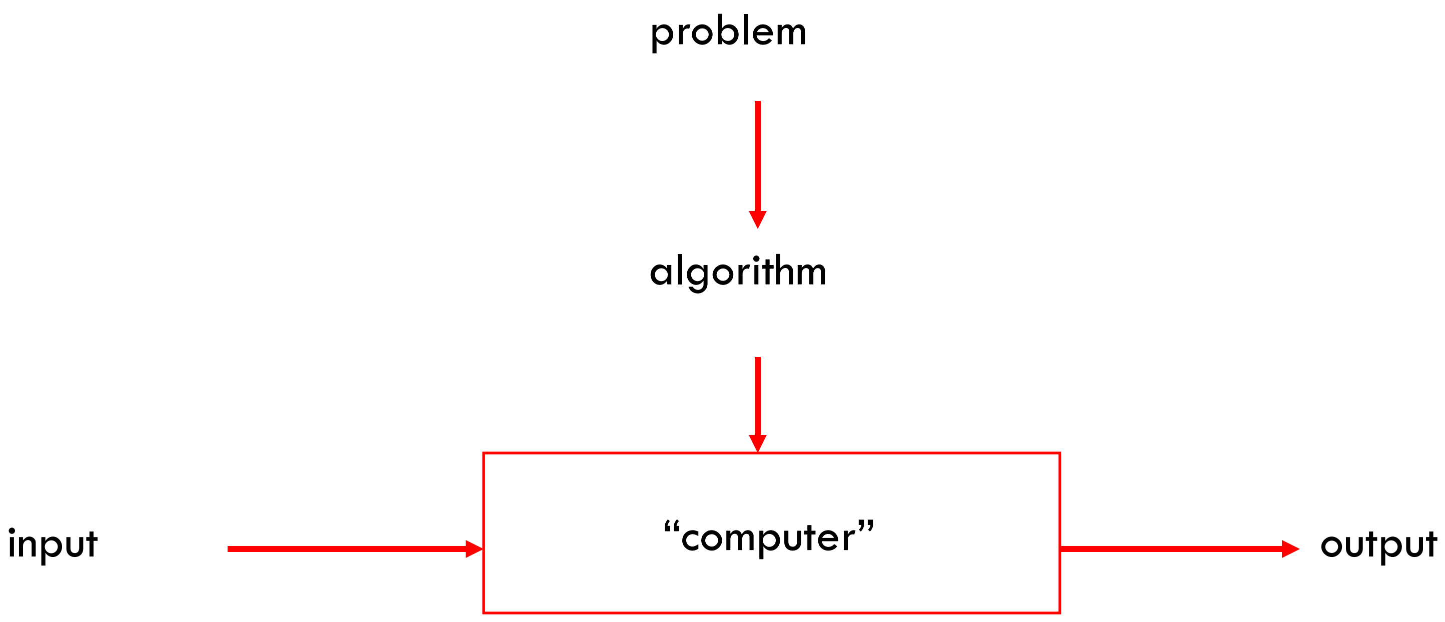
UNIT1

Introduction

Algorithm

- An *Algorithm* is a sequence of unambiguous instructions for solving a problem,
- i.e., for obtaining a required output for any legitimate input in a finite amount of time.

Notion of algorithm



Algorithmic solution

PSEUDOCODE

- Pseudocode (pronounced SOO-doh-kohd) is a detailed yet readable description of what a computer program or algorithm must do, expressed in a formally-styled natural language rather than in a programming language.

- It is sometimes used as a detailed step in the process of developing a program.

- It allows programmers to express the design in great detail and provides programmers a detailed template for the next step of writing code in a specific programming language.

Formatting and Conventions in Pseudocoding

- INDENTATION in pseudocode should be identical to its implementation in a programming language. Try to indent at least four spaces.
- The pseudocode entries are to be cryptic, AND SHOULD NOT BE PROSE. NO SENTENCES.
- No flower boxes in pseudocode.
- Do not include data declarations in pseudocode.

Some Keywords That Should be Used

- For looping and selection,

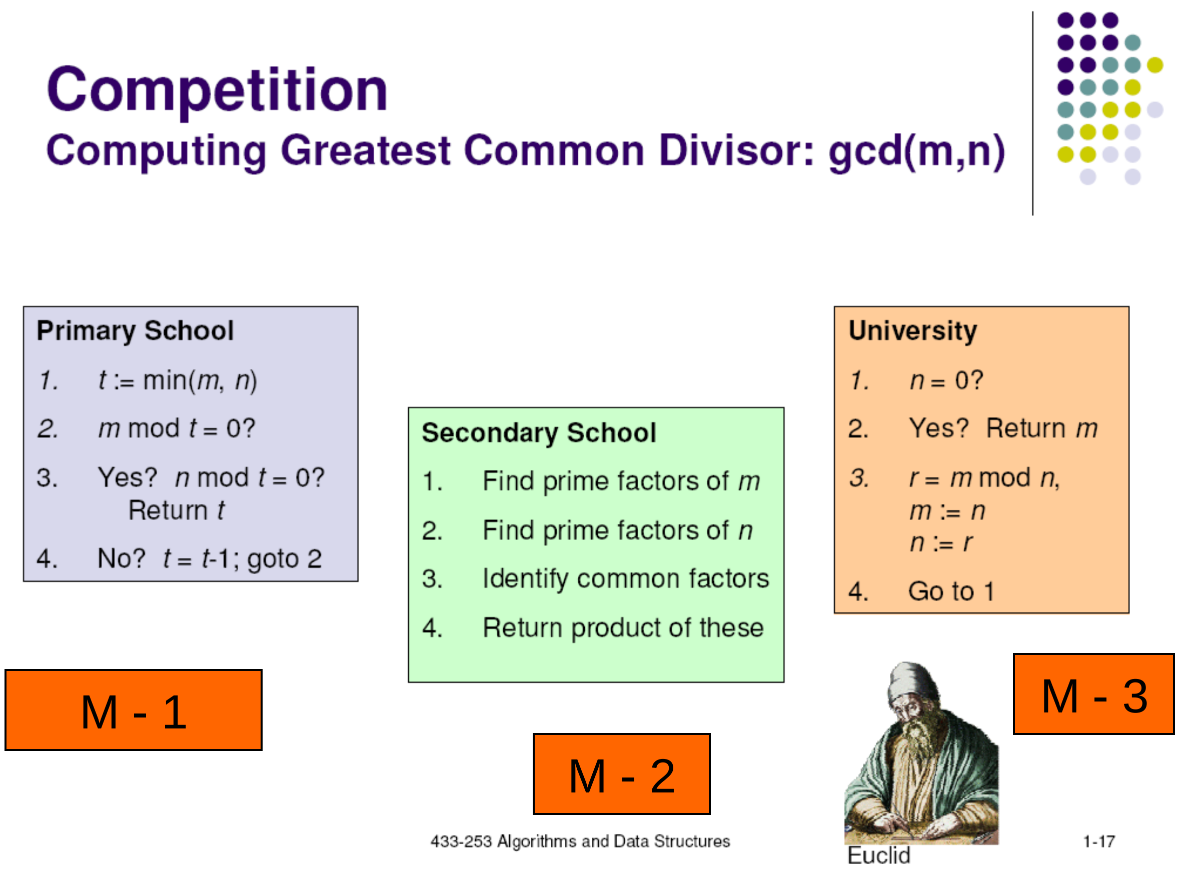
  Do While...EndDo;

  - Do Until...Enddo;
  - Case...EndCase;
  - If...Endif;
  - Call ... with (parameters); Call; Return ....; Return; When; Always use scope terminators for loops and iteration.

Some Keywords …

- As verbs, use the words
  - generate, Compute, Process,
  - Set, reset,
  - increment,
  - calculate,
  - add, sum, multiply, ...
  - print, display,
  - input, output, edit, test , etc.

Methods of finding GCD



Competition

Computing Greatest Common Divisor: gcd(m,n)

Primary School

```
1.  t := min(m, n)
2.  m mod t = 0?
3.  Yes?  n mod t = 0?
        Return t
4.  No?  t = t-1; goto 2
```

M - 1

Secondary School

```
1.  Find prime factors of m
2.  Find prime factors of n
3.  Identify common factors
4.  Return product of these
```

M - 2

University

```
1.  n = 0?
2.  Yes?  Return m
3.  r = m mod n,
    m := n
    n := r
4.  Go to 1
```

Euclid

M - 3

Fundamentals of Analysis of algorithm efficiency

Analysis of algorithms

- Issues:
  - correctness
  - time efficiency
  - space efficiency
  - optimality

- Approaches:
  - theoretical analysis
  - empirical analysis

Theoretical analysis of time efficiency

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

- *Basic operation*: the operation that contributes the most towards the running time of the algorithm

input size

$$T(n) \approx c_{op} C(n)$$

running time

execution time for basic operation or cost

Number of times basic operation is executed

Note: Different basic operations may cost differently!

Input size and basic operation examples

Problem	*Input size measure*	*Basic operation*
Searching for key in a list of *n* items	**Number of list’s items, i.e. *n***	**Key comparison**
Multiplication of two matrices	**Matrix dimensions or total number of elements**	**Multiplication of two numbers**
Checking primality of a given integer *n*	***n*’size = number of digits (in binary representation)**	**Division**
Typical graph problem	**#vertices and/or edges**	**Visiting a vertex or traversing an edge**

Empirical analysis of time efficiency

- Select a specific (typical) sample of inputs
- Use physical unit of time (e.g., milliseconds)

  or

  Count actual number of basic operation's executions
- Analyze the empirical data

Efficiencies

- Worst Case Efficiency:
  - Is its efficiency for the worst case input of size n, which is an input of size n for which the algorithm runs the longest among all possible inputs of that size
  - $C_{worst}(n)$
- Best-case efficiency:
  - Is its efficiency for the worst case input of size n, which is an input of size n for which the algorithm runs the fastest among all possible inputs of that size
  - $C_{best}(n)$

Amortized efficiency

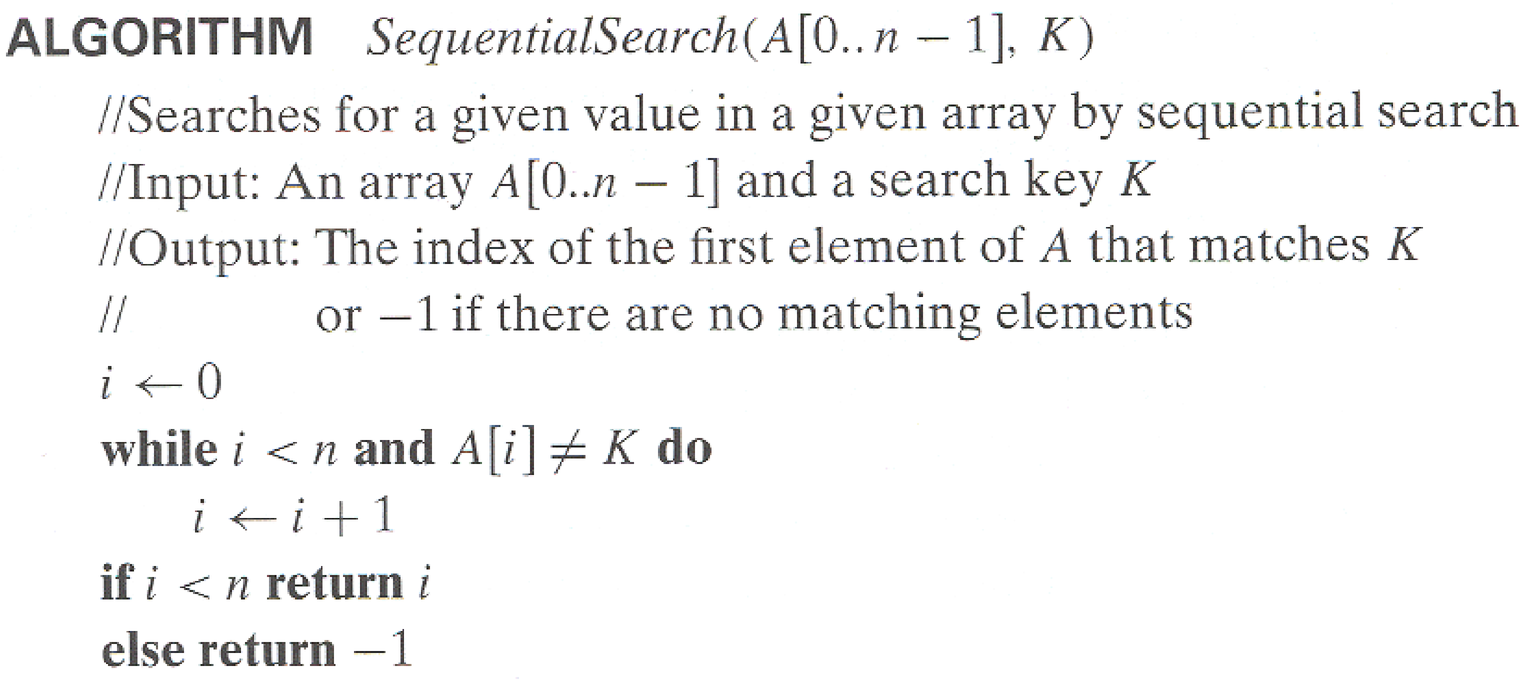
- It applies not to a single run of an algorithm, but rather to a sequence of operations performed on the same data structure

Best-case, average-case, worst-case

For some algorithms, efficiency depends on form of input:

- Worst case: $C_{worst}(n)$ – maximum over inputs of size n
- Best case: $C_{best}(n)$ – minimum over inputs of size n
- Average case: $C_{avg}(n)$ – "average" over inputs of size n
    - Number of times the basic operation will be executed on typical input
    - NOT the average of worst and best case
    - Expected number of basic operations considered as a random variable under some assumption about the probability distribution of all possible inputs. So, avg = expected under uniform distribution.

Example: 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
```

- Worst case — n key comparisons
- Best case — 1 comparisons
- Average case — (n+1)/2, assuming K is in A

Types of formulas for basic operation's count

- Exact formula

  e.g., $C(n) = n(n-1)/2$

- Formula indicating order of growth with specific multiplicative constant

  e.g., $C(n) \approx 0.5\, n^2$

- Formula indicating order of growth with unknown multiplicative constant

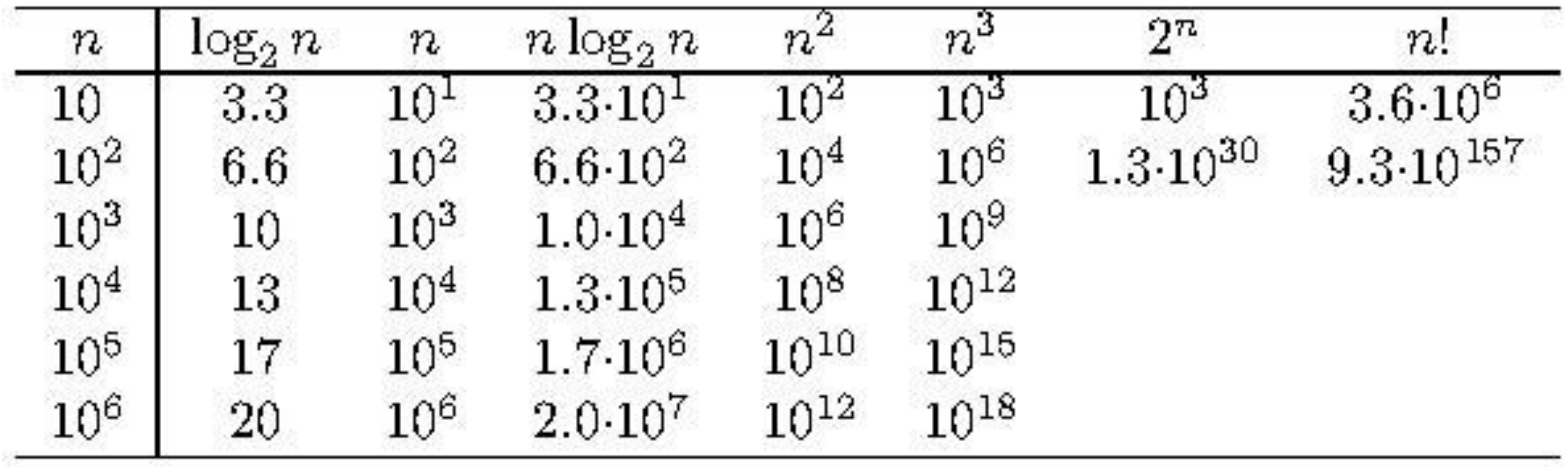
  e.g., $C(n) \approx cn^2$

Order of growth

- Most important: Order of growth within a constant multiple as $n \to \infty$

- Example:
  - How much faster will algorithm run on computer that is twice as fast?
  - How much longer does it take to solve problem of double input size?

Values of some important functions as $n \to \infty$



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}		

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

Asymptotic Notations

- O (Big-Oh)-notation
- Ω (Big-Omega) -notation
- Θ (Big-Theta) -notation

Asymptotic order of growth

A way of comparing functions that ignores constant factors and small input sizes (because?)

- O(*g*(*n*)): class of functions *f*(*n*) that grow <u>no faster</u> than *g*(*n*)

- Θ(*g*(*n*)): class of functions *f*(*n*) that grow <u>at same rate</u> as *g*(*n*)

- Ω(*g*(*n*)): class of functions *f*(*n*) that grow <u>at least as fast</u> as *g*(*n*)

O-notation

Definition: *A function t*(*n*) is said to be in O(*g*(*n*)), denoted t*(n)* $\in$ *O(g(n)) is bounded above by some constant multiple of g(n) for all large n,* i.e., there exist positive constant *c* and non-negative integer n_0 such that

$$f(n) \le c\, g(n) \text{ for every } n \ge n_0$$

Big-oh

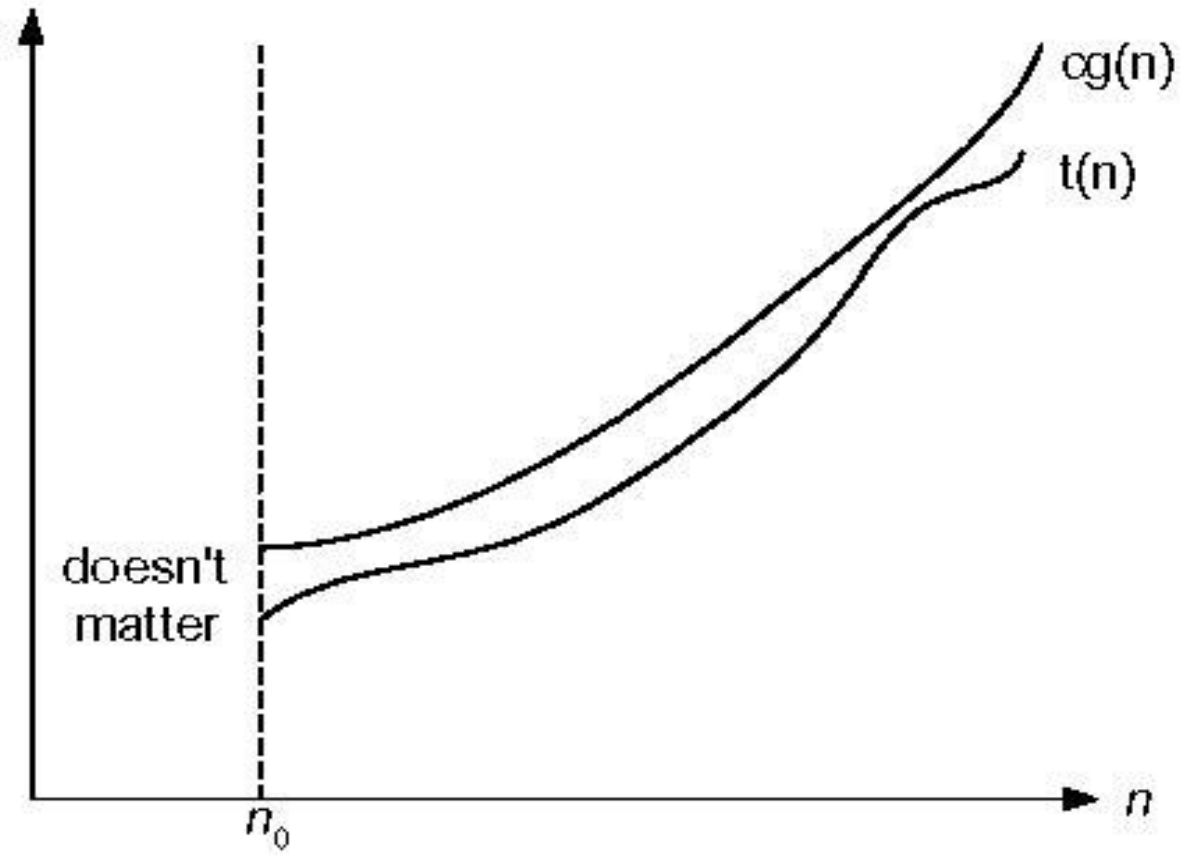


Figure 2.1 Big-oh notation: $t(n) \in O(g(n))$

Ω-notation

- Formal definition
  - A function *t(n)* is said to be in Ω*(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 $n \geq n_0$

Big-omega

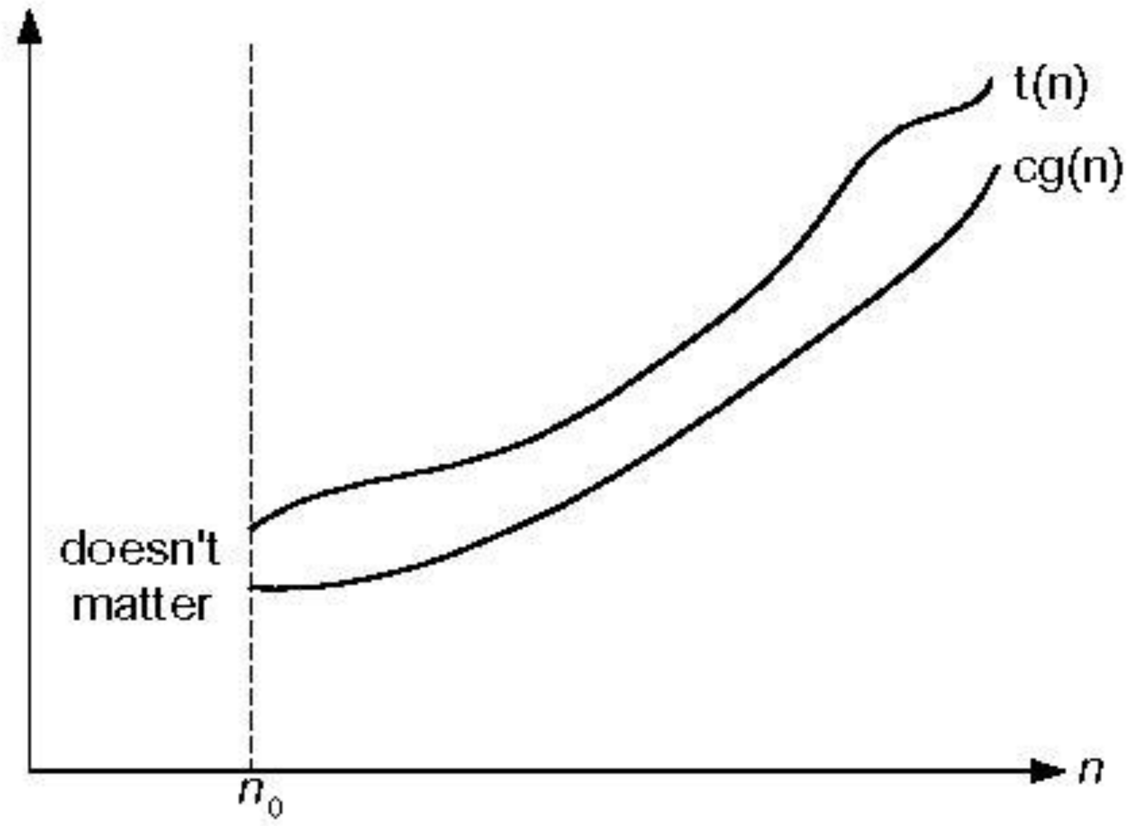


Fig. 2.2 Big-omega notation: $t(n) \in \Omega(g(n))$

Θ-notation

- Formal definition
  - A function *t(n)* is said to be in Θ*(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., <u>if there exist some positive constant c_1 and c_2 and some nonnegative integer n_0 such that</u>

    $c_2\, g(n) \le t(n) \le c_1\, g(n)$ for all $n \ge n_0$

Big-theta

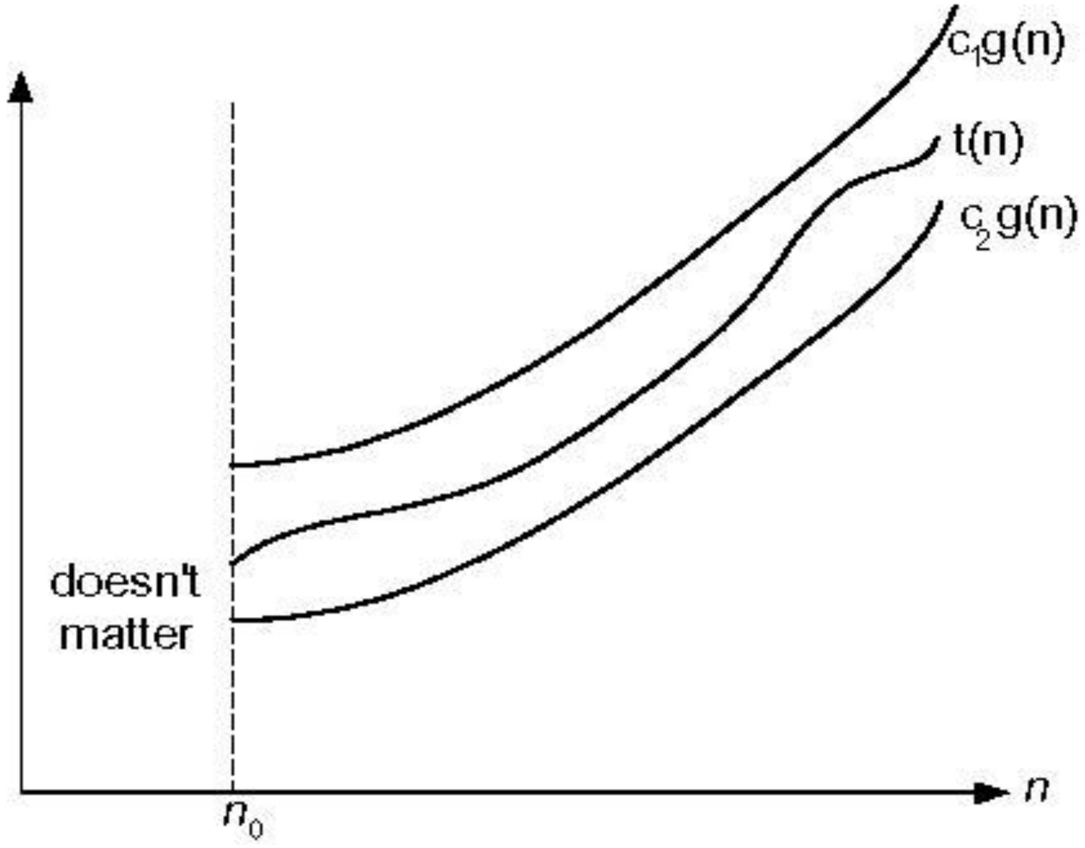


Figure 2.3 Big-theta notation: $t(n) \in \Theta(g(n))$

Theorem

- If $t_1(n) \in O(g_1(n))$ and $t_2(n) \in O(g_2(n))$, then $t_1(n) + t_2(n) \in O(max\{g_1(n), g_2(n)\})$.
  - The analogous assertions are true for the Ω-notation and Θ-notation.

Proof. There exist constants c_1, c_2, n_1, n_2 such that

$$t_1(n) \le c_1 * g_1(n), \quad \text{for all } n \ge n_1$$

$$t_2(n) \le c_2 * g_2(n), \quad \text{for all } n \ge n_2$$

Define $c_3 = c_1 + c_2$ and $n_3 = max\{n_1, n_2\}$. *Then*

$$t_1(n) + t_2(n) \le c_3 * max\{g_1(n), g_2(n)\}, \text{ for all } n \ge n_3$$

Some properties of asymptotic order of growth

- $f(n) \in O(f(n))$
- $f(n) \in O(g(n))$ iff $g(n) \in \Omega(f(\mathrm{n}))$
- If $f(n) \in O(g(n))$ and $g(n) \in O(h(n))$, then $f(n) \in O(h(n))$

  Note similarity with $a \leq \mathrm{b}$
- If $f_1(n) \in O(g_1(n))$ and $f_2(n) \in O(g_2(n))$, then
  $f_1(n) + f_2(n) \in O(\max\{g_1(n), g_2(n)\})$

Also, $\Sigma_{1\leq i\leq n} \Theta(f(i)) = \Theta\ (\Sigma_{1\leq i\leq n} f(i))$

Establishing order of growth using limits

$$\lim_{n\to\infty} T(n)/g(n) = \begin{cases} 0 & \text{order of growth of } T(n) < \text{ order of growth of } g(n) \\ c > 0 & \text{order of growth of } T(n) = \text{ order of growth of } g(n) \\ \infty & \text{order of growth of } T(n) > \text{ order of growth of } g(n) \end{cases}$$

L’Hôpital’s rule and Stirling’s formula

L’Hôpital’s rule: If $\lim_{n\to\infty} f(n) = \lim_{n\to\infty} g(n) = \infty$ and the derivatives f', g' exist, then

$$\lim_{n\to\infty} \frac{f(n)}{g(n)} = \lim_{n\to\infty} \frac{f'(n)}{g'(n)}$$

Example: log n vs. n

Stirling’s formula: $n! \approx (2\pi n)^{1/2} (n/e)^n$

Example: 2^n vs. $n!$

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

  because $$\log_a n = \log_b n / \log_b a$$

- 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 a^n have different orders of growth for different a's

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

Basic asymptotic 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

Plan for analyzing nonrecursive algorithms

General Plan for Analysis

- Decide on parameter *n* indicating *input size*
- Identify algorithm's *basiyc operation*
- Determine *worst*, *average*, and *best* cases for input of size *n*
- Set up a sum for the number of times the basic operation is executed
- Simplify the sum using standard formulas and rules (see Appendix A)

Useful summation formulas and rules

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

In particular, $\Sigma_{1 \le i \le n} 1 = n - 1 + 1 = n \in \Theta(n)$

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

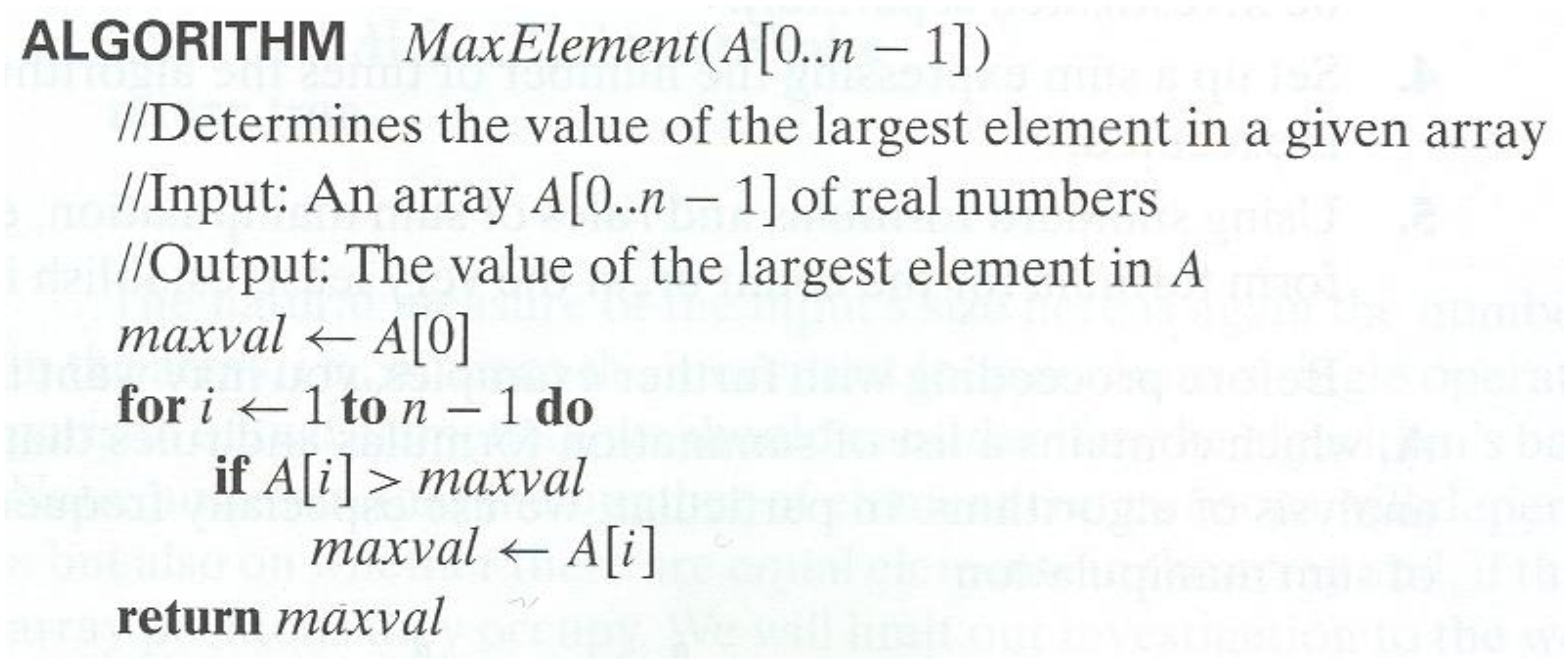
$\Sigma_{1 \le i \le n} i^2 = 1^2+2^2+\ldots+n^2 = n(n+1)(2n+1)/6 \approx n^3/3 \in \Theta(n^3)$

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

In particular, $\Sigma_{0 \le i \le n} 2^i = 2^0 + 2^1 + \ldots + 2^n = 2^{n+1} - 1 \in \Theta(2^n)$

$\Sigma(a_i \pm b_i) = \Sigma a_i \pm \Sigma b_i$ $\quad \Sigma c a_i = c \Sigma a_i$ $\quad \Sigma_{l \le i \le u} a_i = \Sigma_{l \le i \le m} a_i + \Sigma_{m+1 \le i \le u} a_i$

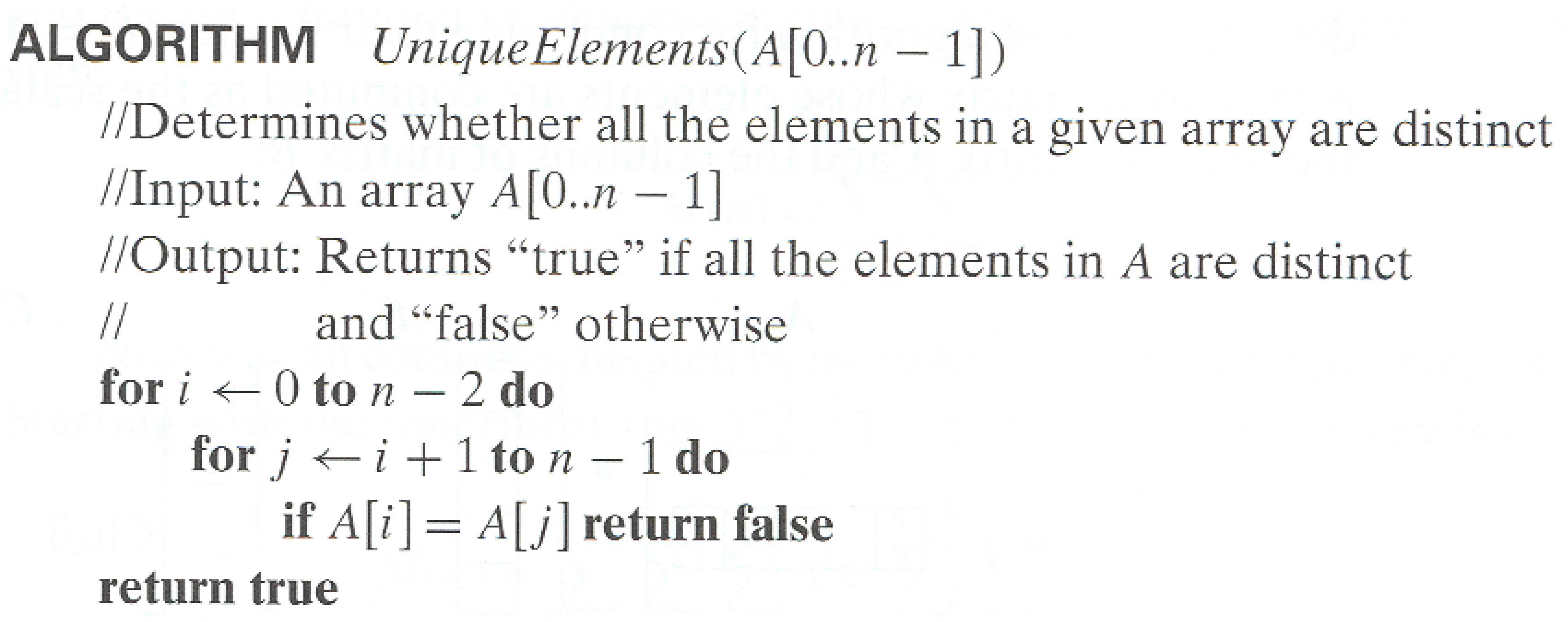
Example 1: Maximum element



```
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 $A$
    $maxval \leftarrow A[0]$
    for $i \leftarrow 1$ to $n-1$ do
        if $A[i] > maxval$
            $maxval \leftarrow A[i]$
    return $maxval$
```

T(n) = $\sum_{1 \le i \le n-1} 1 = n-1 = \Theta(n)$ comparisons

Example 2: Element uniqueness problem



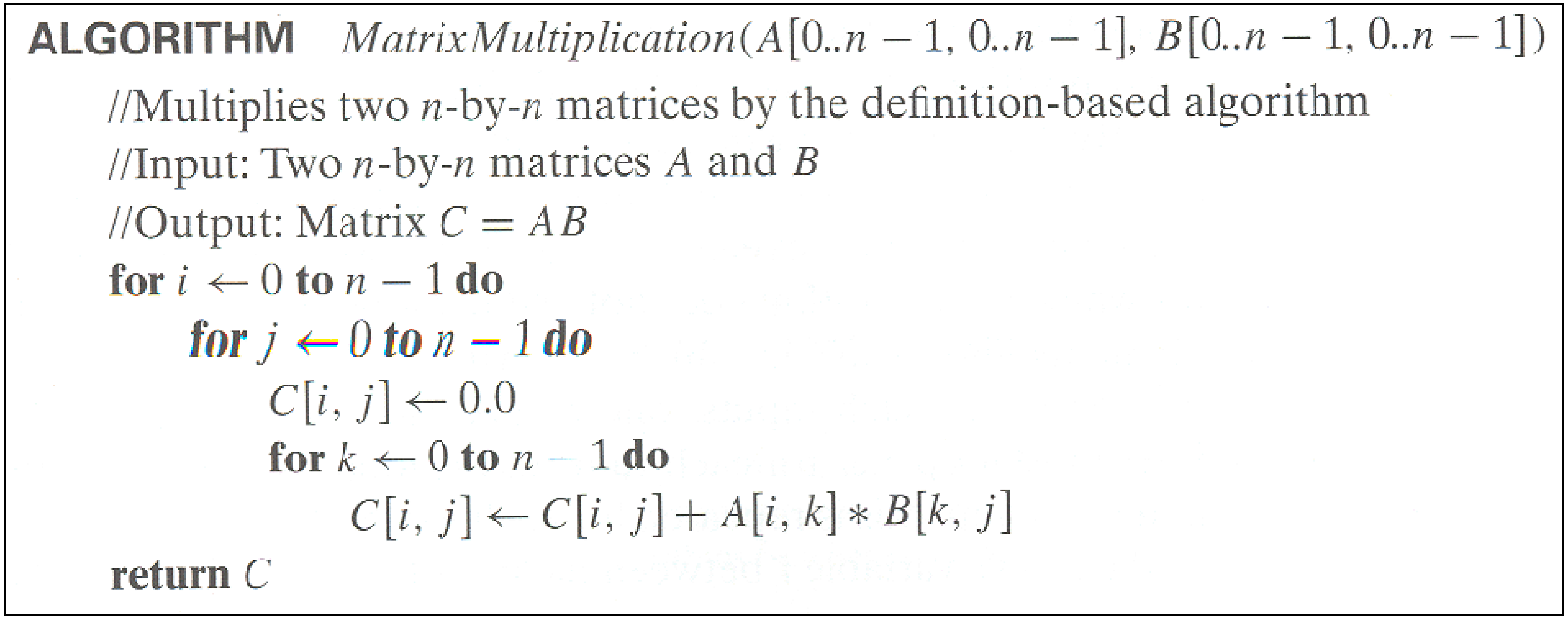
```
ALGORITHM  UniqueElements(A[0..n − 1])
    //Determines whether all the elements in a given array are distinct
    //Input: An array A[0..n − 1]
    //Output: Returns “true” if all the elements in A are distinct
    //              and “false” otherwise
    for $i \leftarrow 0$ to $n - 2$ do
        for $j \leftarrow i + 1$ to $n - 1$ do
            if $A[i] = A[j]$ return false
    return true
```

$$T(n) = \Sigma_{0 \le i \le n-2} \left(\Sigma_{i+1 \le j \le n-1} 1\right)$$

$$= \Sigma_{0 \le i \le n-2}\, n-i-1 = (n-1+1)(n-1)/2$$

$$= \Theta(n^2) \text{ comparisons}$$

Example 3: Matrix multiplication



```
ALGORITHM  $MatrixMultiplication(A[0..n-1, 0..n-1], B[0..n-1, 0..n-1])$
    //Multiplies two $n$-by-$n$ matrices by the definition-based algorithm
    //Input: Two $n$-by-$n$ matrices $A$ and $B$
    //Output: Matrix $C = AB$
    for $i \leftarrow 0$ to $n-1$ do
        for $j \leftarrow 0$ to $n-1$ do
            $C[i, j] \leftarrow 0.0$
            for $k \leftarrow 0$ to $n-1$ do
                $C[i, j] \leftarrow C[i, j] + A[i, k] * B[k, j]$
    return $C$
```

$T(n) = \Sigma_{0 \le i \le n-1} \Sigma_{0 \le i \le n-1}\ n$

$= \Sigma_{0 \le i \le n-1} \Theta(n^2)$

$= \Theta(n^3)$ multiplications

Example 4: Gaussian elimination

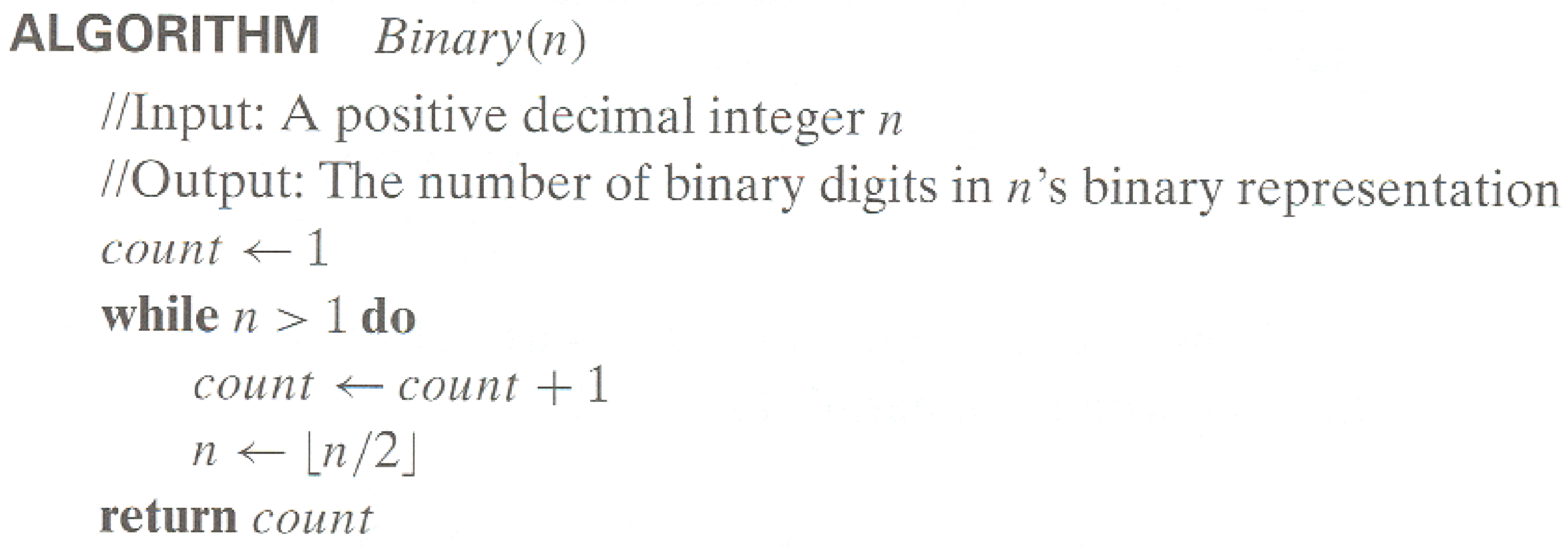
Algorithm *GaussianElimination*(*A*[0..*n*-1,0..*n*])
//Implements Gaussian elimination on an *n*-by-(*n*+1) matrix *A*

```
for $i \leftarrow 0$ to $n - 2$ do
    for $j \leftarrow i + 1$ to $n - 1$ do
        for $k \leftarrow i$ to $n$ do
            $A[j,k] \leftarrow A[j,k] - A[i,k] * A[j,i] / A[i,i]$
```

Find the efficiency class and a constant factor improvement.

```
for $i \leftarrow 0$ to $n - 2$ do
    for $j \leftarrow i + 1$ to $n - 1$ do
        $B \leftarrow A[j,i] / A[i,i]$
        for $k \leftarrow i$ to $n$ do
            $A[j,k] \leftarrow A[j,k] - A[i,k] * B$
```

Example 5: Counting binary digits



```
ALGORITHM Binary(n)
    //Input: A positive decimal integer n
    //Output: The number of binary digits in n's binary representation
    count ← 1
    while n > 1 do
        count ← count + 1
        n ← ⌊n/2⌋
    return count
```

Plan for Analysis of Recursive Algorithms

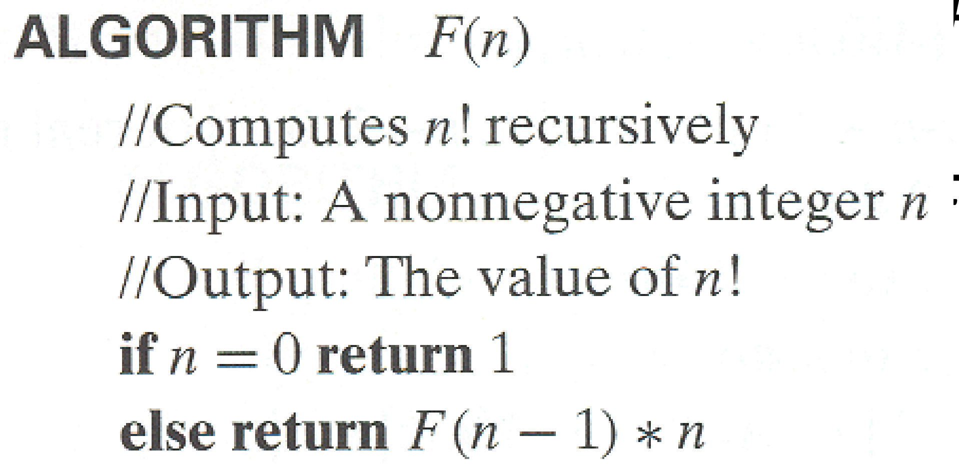
- Decide on a parameter indicating an input’s size.
- Identify the algorithm’s basic operation.
- Check whether the number of times the basic op. is executed may vary on different inputs of the same size. (If it may, the worst, average, and best cases must be investigated separately.)
- Set up a recurrence relation with an appropriate initial condition expressing the number of times the basic op. is executed.
- Solve the recurrence (or, at the very least, establish its solution’s order of growth) by backward substitutions or another method.

Example 1: Recursive evaluation of *n*!

Definition: $n\,! = 1 * 2 * \ldots *(n\text{-}1) * n$ for $n \geq 1$ and $0! = 1$

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

$F(0) = 1$



```
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
```

n

multiplication

M(n) = M(n-1) + 1

M(0) = 0

Size:

Solving the recurrence for M(n)

$M(n) = M(n\text{-}1) + 1,\ \ M(0) = 0$

$M(n) = M(n\text{-}1) + 1$

$= (M(n\text{-}2) + 1) + 1 \quad = \quad M(n\text{-}2) + 2$

$= (M(n\text{-}3) + 1) + 2 \quad = \quad M(n\text{-}3) + 3$

…

$= M(n\text{-}i) + i$

$= M(0) + n$

$= n$

The method is called backward substitution.

Solving recurrence for number of moves

$M(n) = 2M(n\text{-}1) + 1, \quad M(1) = 1$

M(n) = 2M(n-1) + 1

= 2(2M(n-2) + 1) + 1 = 2^2*M(n-2) + 2^1 + 2^0

= 2^2*(2M(n-3) + 1) + 2^1 + 2^0

= 2^3*M(n-3) + 2^2 + 2^1 + 2^0

= …

= 2^(n-1)*M(1) + 2^(n-2) + … + 2^1 + 2^0

= 2^(n-1) + 2^(n-2) + … + 2^1 + 2^0

= 2^n - 1

DIVIDE AND CONQUER

Divide and Conquer

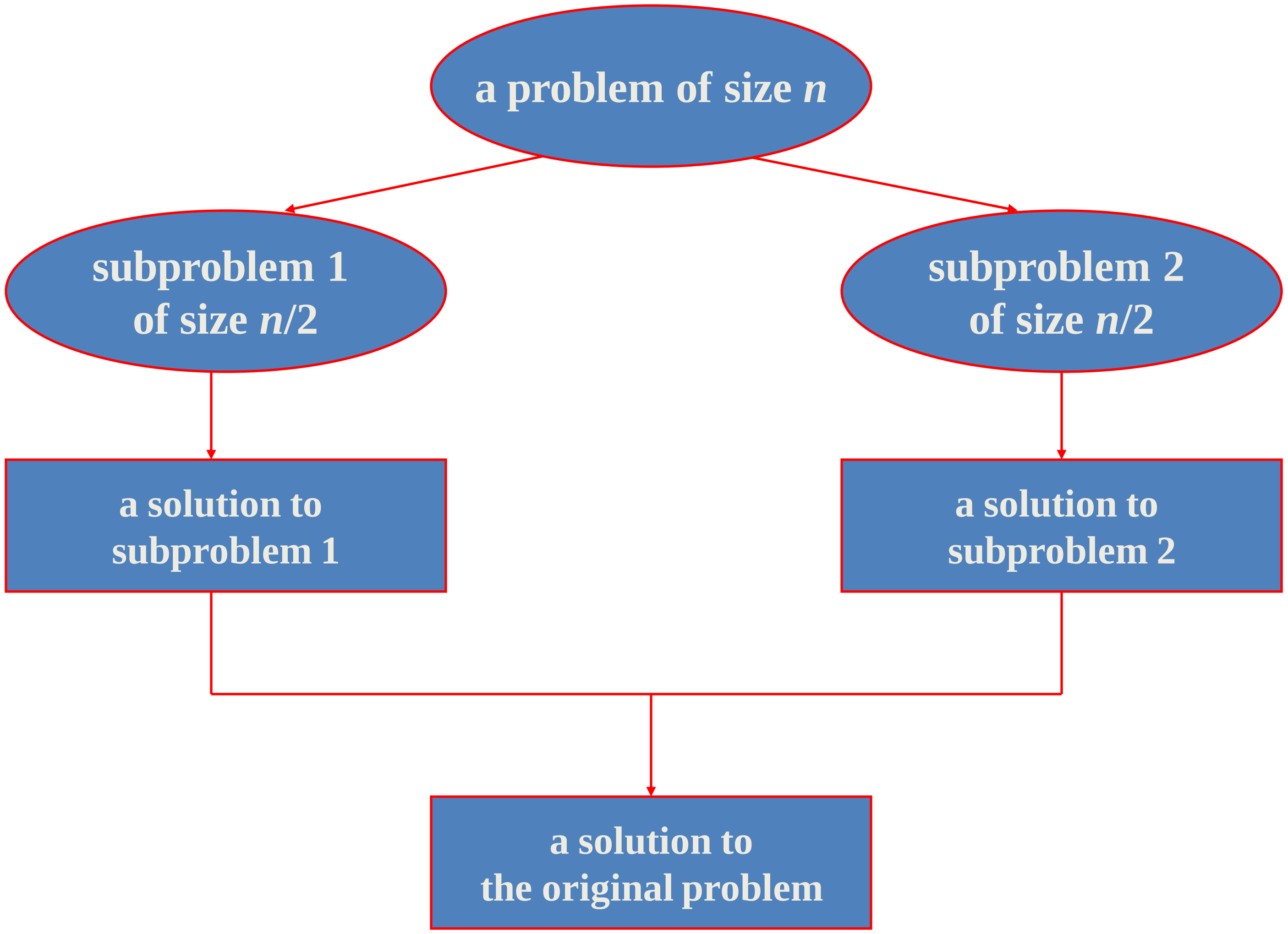
The most well known algorithm design strategy:

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

2. Solve smaller instances recursively

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

Divide-and-conquer technique


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

Divide and Conquer Examples

- ☐ Sorting: mergesort and quicksort
- ☐ Tree traversals
- ☐ Binary search
- ☐ Matrix multiplication-Strassen’s algorithm
- ☐ Convex hull-QuickHull algorithm

General Divide and Conquer recurrence: Master Theorem

$T(n) = aT(n/b) + f(n)$ where $f(n) \in \Theta(n^d)$

1. $a < b^d$ $T(n) \in \Theta(n^d)$
2. $a = b^d$ $T(n) \in \Theta(n^d \lg n)$
3. $a > b^d$ $T(n) \in \Theta(n^{\log_b a})$

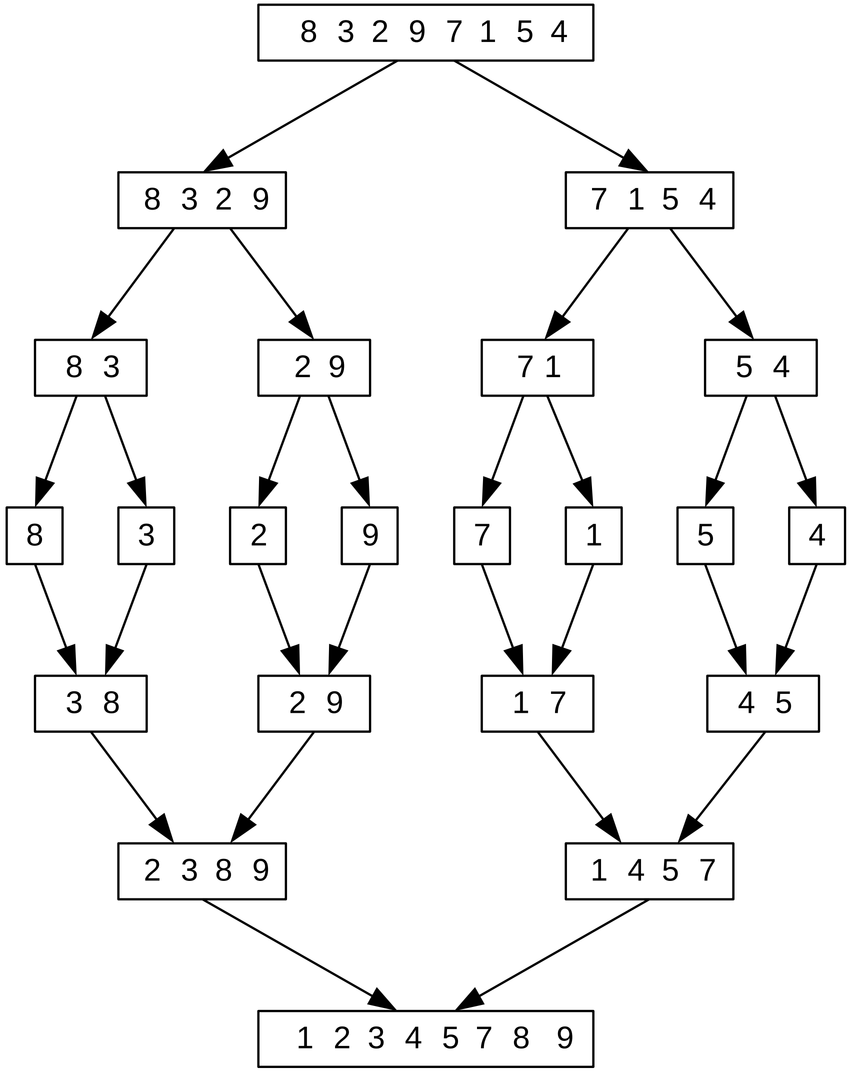
Note: the same results hold with O instead of Θ.

Mergesort

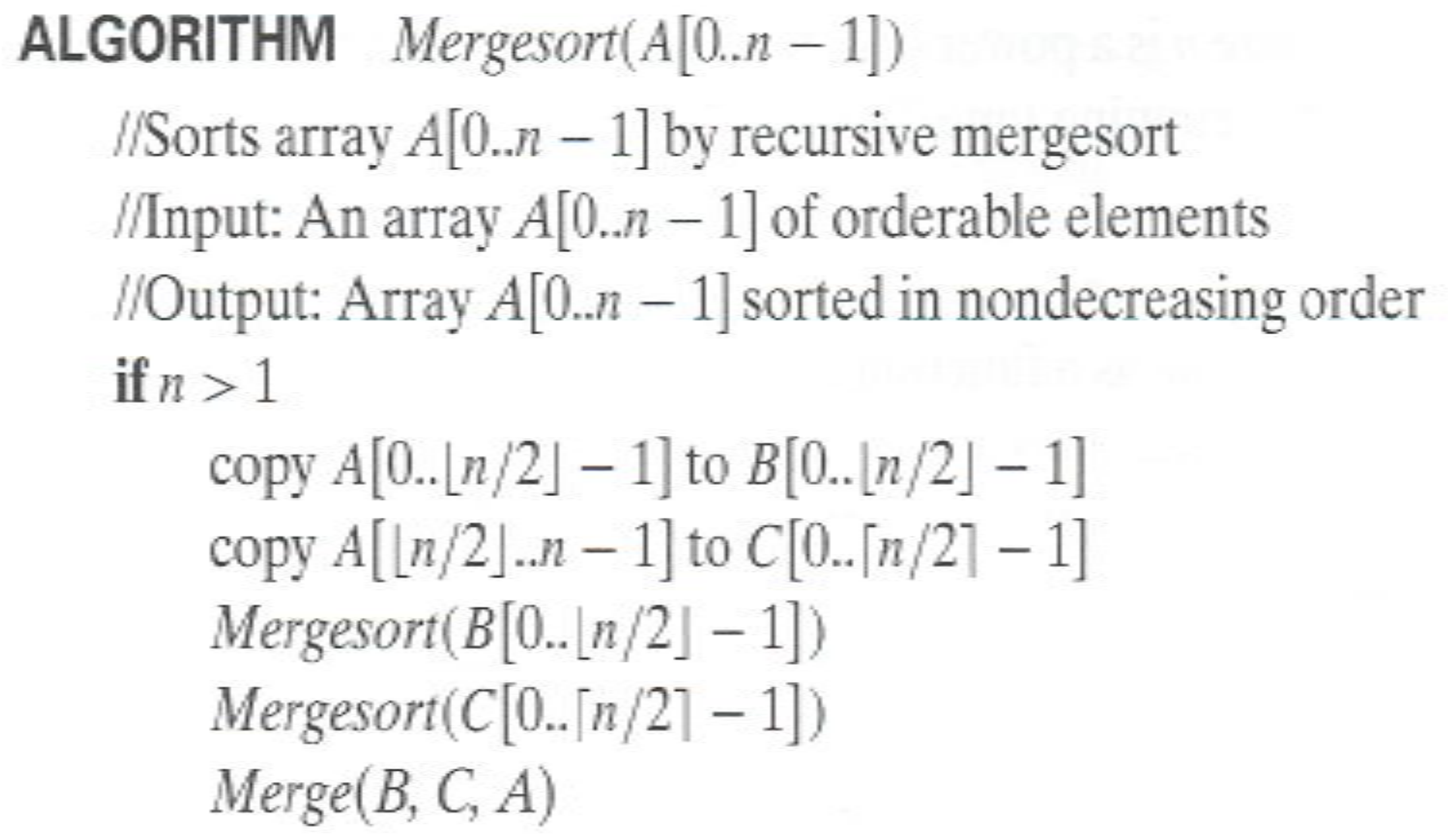
Algorithm:

- ☐ Split array A[1..*n*] in two and make copies of each half in arrays B[1.. $\lfloor n/2 \rfloor$] and C[1.. $\lceil n/2 \rceil$]
- ☐ Sort arrays B and C
- ☐ 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.

Mergesort Example

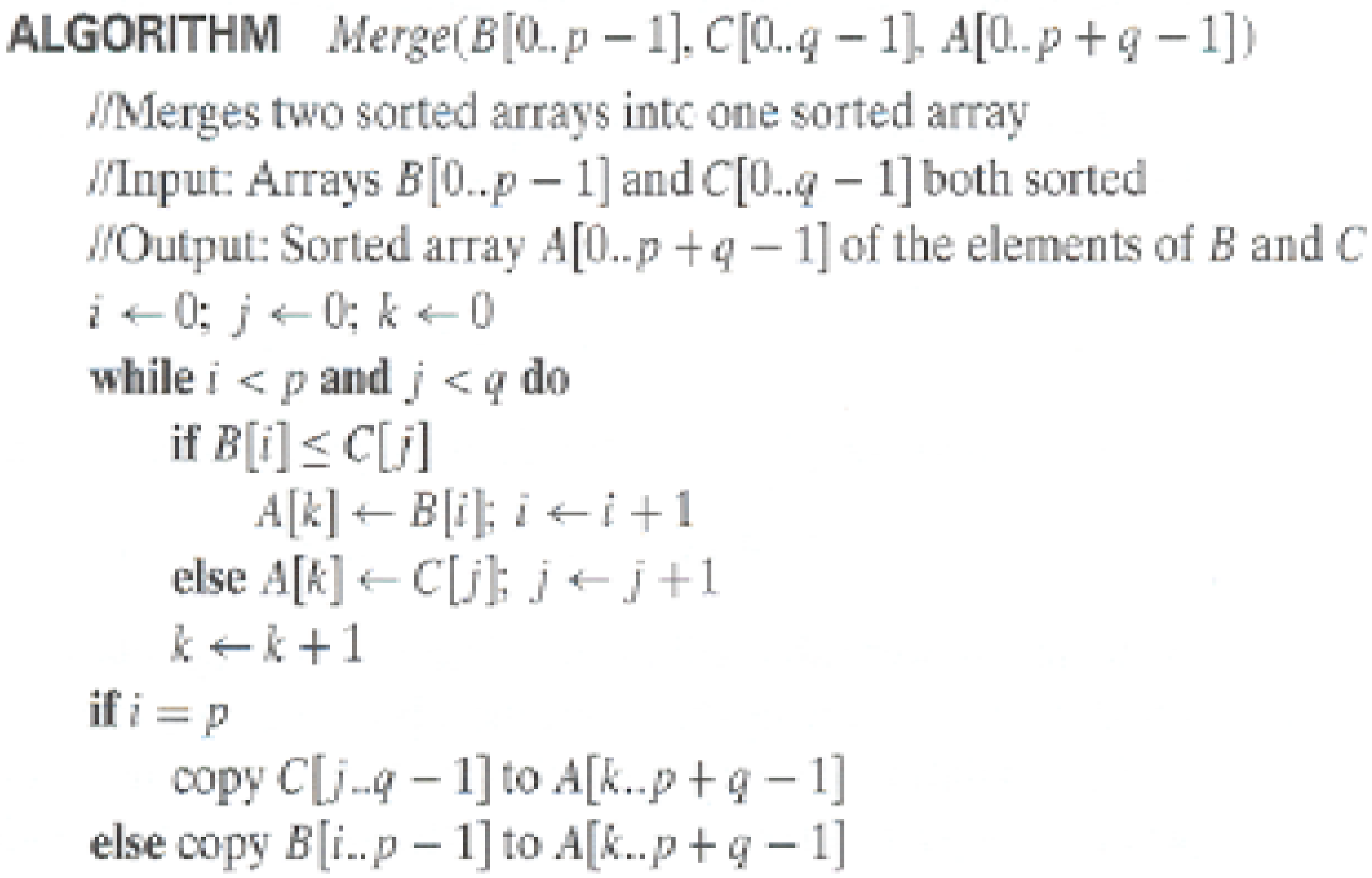


Pseudocode for Mergesort



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

Pseudocode for Merge



```
ALGORITHM  Merge(B[0..p − 1], C[0..q − 1], A[0..p + q − 1])
    //Merges two sorted arrays intc one sorted array
    //Input: Arrays B[0..p − 1] and C[0..q − 1] both sorted
    //Output: Sorted array A[0..p + q − 1] of the elements of B and C
    $i \leftarrow 0$;  $j \leftarrow 0$;  $k \leftarrow 0$
    while $i < p$ and $j < q$ do
        if $B[i] \le C[j]$
            $A[k] \leftarrow B[i]$;  $i \leftarrow i + 1$
        else $A[k] \leftarrow C[j]$;  $j \leftarrow j + 1$
        $k \leftarrow k + 1$
    if $i = p$
        copy $C[j..q - 1]$ to $A[k..p + q - 1]$
    else copy $B[i..p - 1]$ to $A[k..p + q - 1]$
```

Recurrence Relation for Mergesort

- Let $T(n)$ be worst case time on a sequence of n keys
- If $n = 1$, then $T(n) = \Theta(1)$ (constant)
- If $n > 1$, then $T(n) = 2\ T(n/2) + \Theta(n)$
  - two subproblems of size $n/2$ each that are solved recursively
  - $\Theta(n)$ time to do the merge

Efficiency of mergesort

- All cases have same efficiency: Θ(n log n)

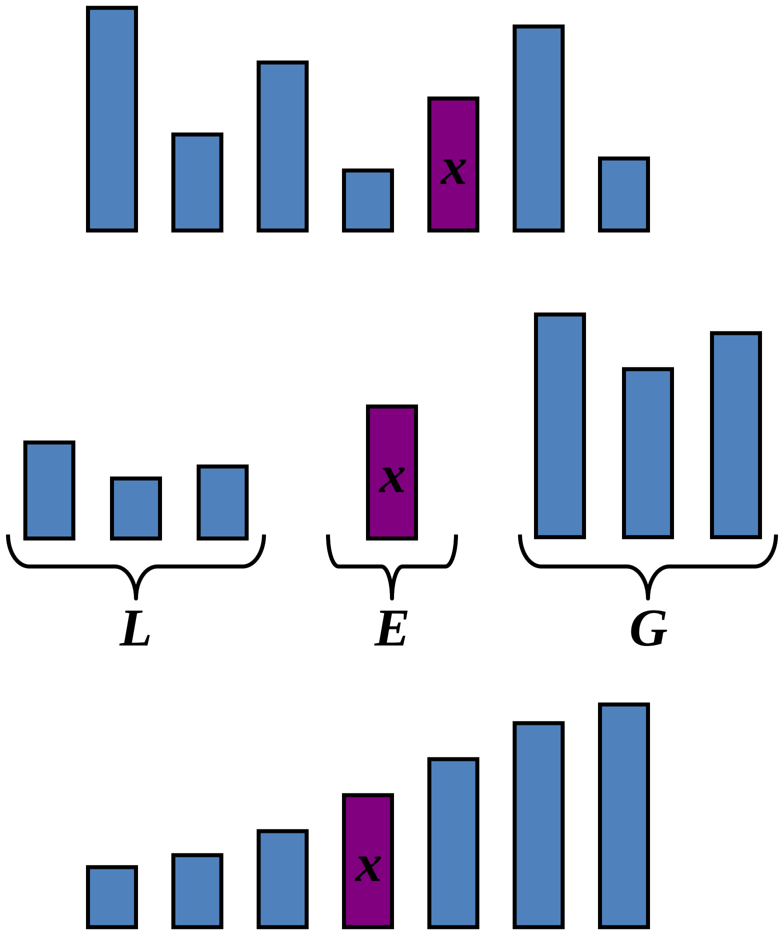
- Number of comparisons is close to theoretical minimum for comparison-based sorting:
  - log n ! ≈ n lg n - 1.44 n

- Space requirement: Θ(n) (<u>NOT</u> in-place)

- Can be implemented without recursion (bottom-up)

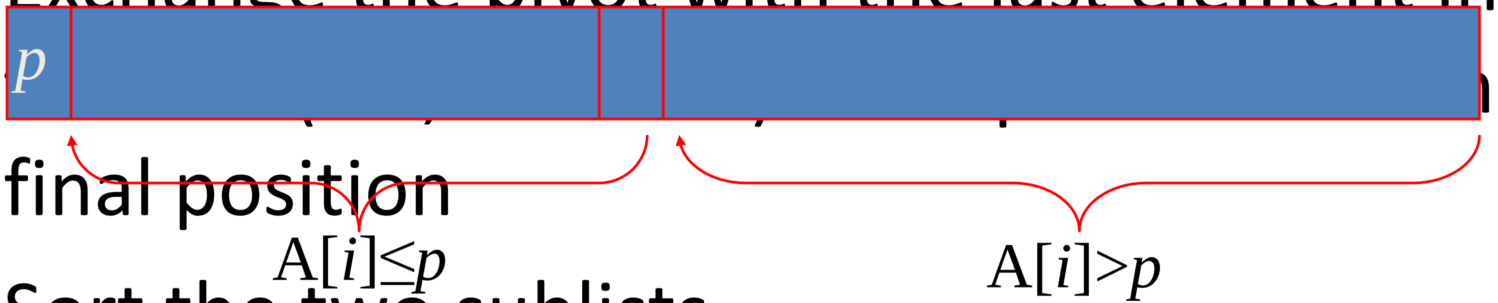
Quick-Sort

- Quick-sort is a randomized sorting algorithm based on the divide-and-conquer paradigm:
- Divide: pick a random element *x* (called pivot) and partition *S* into
  - *L* elements less than *x*
  - *E* elements equal *x*
  - *G* elements greater than *x*
- Recur: sort *L* and *G*
- Conquer: join *L*, *E* and *G*



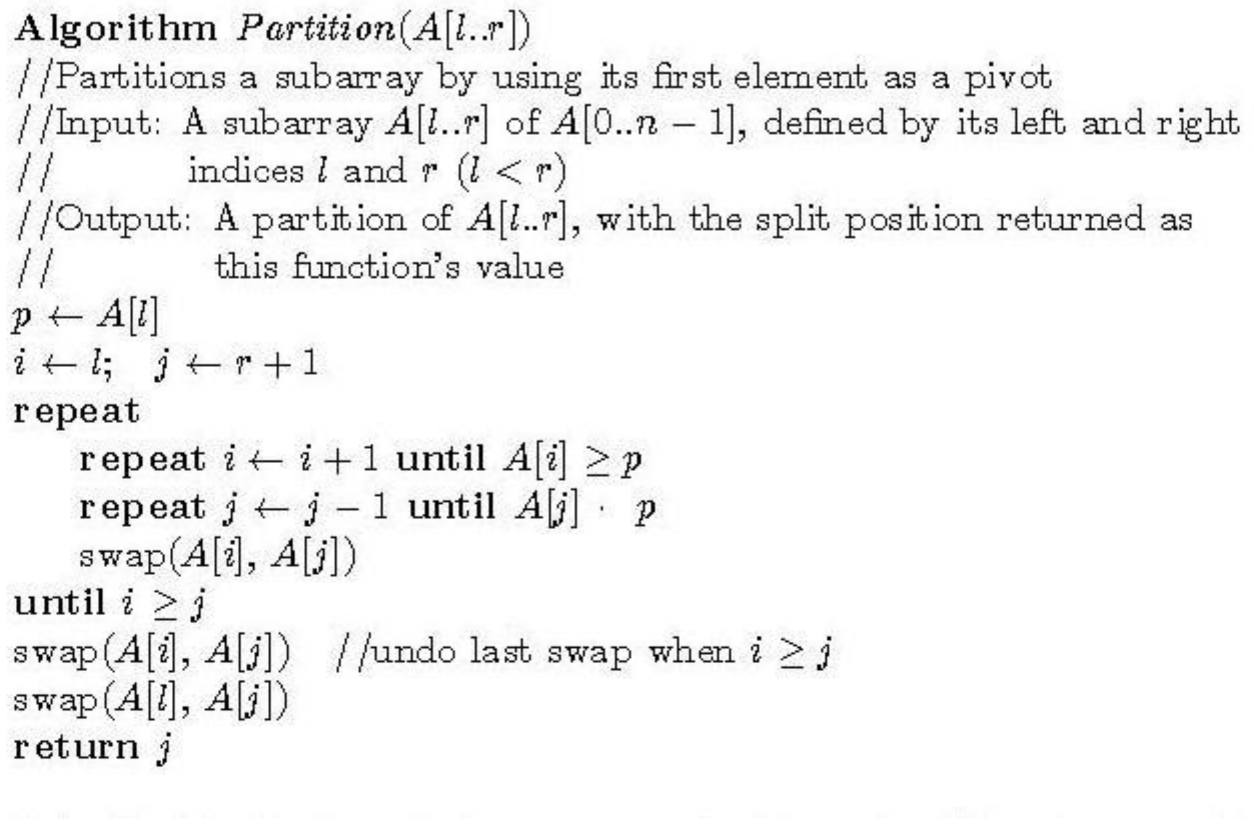
Quicksort

- Select a *pivot* (partitioning element)
- Rearrange the list so that all the elements in the positions before the pivot are smaller than or equal to the pivot and those after the pivot are larger than the pivot
- Exchange the pivot with the last element in its final position



- Sort the two sublists

The partition 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
$p \leftarrow A[l]$
$i \leftarrow l$;   $j \leftarrow r + 1$
repeat
    repeat $i \leftarrow i + 1$ until $A[i] \geq p$
    repeat $j \leftarrow j - 1$ until $A[j] \cdot p$
    swap(A[i], A[j])
until $i \geq j$
swap(A[i], A[j])   //undo last swap when $i \geq j$
swap(A[l], A[j])
return j
```

Efficiency of quicksort

- *Best case*: split in the middle — Θ(n log n)
- *Worst case*: sorted array! — Θ(n^2)
- *Average case*: random arrays — Θ(n log n)

- Improvements:
  - better pivot selection: median of three partitioning avoids worst case in sorted files
  - switch to insertion sort on small subfiles

- Considered the method of choice for internal sorting for large files ($n \geq 10000$)

Binary Search - an Iterative Algorithm

Very efficient algorithm for searching in sorted array:

K vs A[0] . . . A[m] . . . A[n-1]

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

otherwise, continue searching by the same method

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

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

Pseudocode for Binary Search

```
ALGORITHM BinarySearch(A[0..n-1], K)
$l \leftarrow 0$;   $r \leftarrow n-1$
while $l \leq r$ do                      // $l$ and $r$ crosses over → can't find K
    $m \leftarrow \lfloor (l+r)/2 \rfloor$
    if $K = A[m]$ return $m$             //the key is found
    else if $K < A[m]$ $r \leftarrow m-1$    //the key is on the left half of the array
    else $l \leftarrow m+1$                 // the key is on the right half of the array
return -1
```

Binary Search – a Recursive Algorithm

```
ALGORITHM BinarySearchRecur(A[0..n-1], l, r, K)
if l > r
    return –1
else
    m ← $\lfloor (l + r) / 2 \rfloor$
    if K = A[m]
        return m
    else if K < A[m]
        return BinarySearchRecur(A[0..n-1], l, m-1, K)
    else
        return BinarySearchRecur(A[0..n-1], m+1, r, K)
```

Analysis of Binary Search

- <u>Worst-case (successful or fail)</u> :
  - $C_w(n) = 1 + C_w(\lfloor n/2 \rfloor)$,
  - $C_w(1) = 1$

  solution: $C_w(n) = \lfloor \log_2 n \rfloor + 1 = \lceil \log_2(n+1) \rceil$

- This is VERY fast: e.g., $C_w(10^6) = 20$

- <u>Best-case:</u> successful $C_b(n) = 1$,

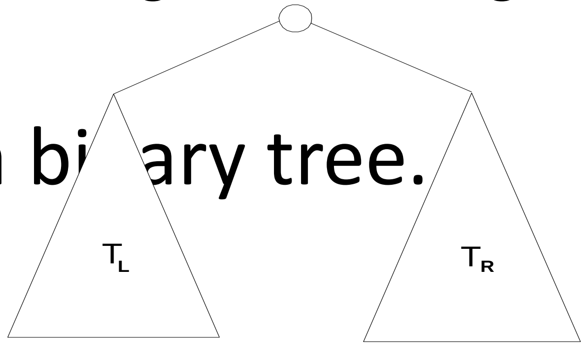
  fail $C_b(n) = \lfloor \log_2 n \rfloor + 1$

- <u>Average-case</u>: successful $C_{avg}(n) = \log_2 n - 1$

  fail $C_{avg}(n) = \log_2(n+1)$

Binary Tree Traversals

- Definitions
  - 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 T_L and T_R called, respectively, the left and right subtree of the root.
  - The height of a tree is defined as the length of the longest path from the root to a leaf.
- Problem: find the height of a binary tree.



Pseudocode - Height of a Binary Tree

```
ALGORITHM Height($T$)
//Computes recursively the height of a binary tree
//Input: A binary tree $T$
//Output: The height of $T$
if $T = \varnothing$
    return –1
else
    return max{Height($T_L$), Height($T_R$)} + 1
```

Analysis:

Number of comparisons of a tree T with $\varnothing$: $2n + 1$

Number of comparisons made to compute height is the same as number of additions:

$A(n(T)) = A(n(T_L)) + A(n(T_R)) + 1$ for $n>0$,

$A(0) = 0$

The solution is $A(n) = n$

Binary Tree Traversals– preorder, inorder, and postorder traversal

- Binary tee traversal: visit all nodes of a binary tree recursively.

```
Algorithm Preorder(T)
//Implement the preorder traversal of a binary tree
//Input: Binary tree T (with labeled vertices)
//Output: Node labels listed in preorder
if T ‡ ∅
        write label of T's root
        Preorder($T_L$)
        Preorder($T_R$)
```

Multiplication of Large Integers

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

A = 12345678901357986429   B = 87654321284820912836

The grade-school algorithm:

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

Efficiency: n^2 one-digit multiplications

First Divide-and-Conquer Algorithm

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

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

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

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

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

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

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

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

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

Second Divide-and-Conquer Algorithm

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

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

$(A_1 + A_2) * (B_1 + B_2) = A_1 * B_1 + (A_1 * B_2 + A_2 * B_1) + A_2 * B_{2,}$

I.e., $(A_1 * B_2 + A_2 * B_1) = (A_1 + A_2) * (B_1 + B_2) - A_1 * B_1 - A_2 * B_{2,}$

which requires only 3 multiplications at the expense of (4-1) extra add/sub.

Recurrence for the number of multiplications M(n):

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

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

Strassen's matrix multiplication

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

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

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

Submatrices:

- $M_1 = (A_{00} + A_{11}) * (B_{00} + B_{11})$
- $M_2 = (A_{10} + A_{11}) * B_{00}$
- $M_3 = A_{00} * (B_{01} - B_{11})$
- $M_4 = A_{11} * (B_{10} - B_{00})$
- $M_5 = (A_{00} + A_{01}) * B_{11}$
- $M_6 = (A_{10} - A_{00}) * (B_{00} + B_{01})$
- $M_7 = (A_{01} - A_{11}) * (B_{10} + B_{11})$

Efficiency of Strassen’s algorithm

- If *n* is not a power of 2, matrices can be padded with zeros

- Number of multiplications: 7

- Number of additions: 18

Time Analysis

$$T(1) = 1 \qquad \text{(assume } N = 2^k\text{)}$$

$$T(N) = 7T(N/2)$$

$$T(N) = 7^k T(N/2^k) = 7^k$$

$$T(N) = 7^{\log N} = N^{\log 7} = N^{2.81}$$

Standard vs Strassen

	N	Multiplications	Additions
Standard alg.	100	1,000,000	990,000
Strassen's alg.	100	411,822	2,470,334
Standard alg.	1000	1,000,000,000	999,000,000
Strassen's alg.	1000	264,280,285	1,579,681,709
Standard alg.	10,000	10^{12}	$9.99*10^{11}$
Strassen's alg.	10,000	$0.169*10^{12}$	10^{12}

Greedy Technique

Greedy Technique

Constructs a solution to an *optimization problem* piece by piece through a sequence of choices that are:

Defined by an objective function and a set of constraints

- feasible, i.e. satisfying the constraints
- locally optimal (with respect to some neighborhood definition)
- greedy (in terms of some measure), and irrevocable

For some problems,

it yields a globally optimal solution for every instance.

For most, does not but can be useful for fast approximations.

Applications of the Greedy Strategy

- Optimal solutions:
  - change making for “normal” coin denominations
  - minimum spanning tree (MST)
  - single-source shortest paths
  - simple scheduling problems
  - Huffman codes

- Approximations/heuristics:
  - traveling salesman problem (TSP)
  - knapsack problem
  - other combinatorial optimization problems

Change-Making Problem

Given unlimited amounts of coins of denominations $d_1 > \ldots > d_m$,

give change for amount n with the least number of coins

Q: What are the objective function and constraints?

Example: $d_1 = 25c$, $d_2 = 10c$, $d_3 = 5c$, $d_4 = 1c$ and $n = 48c$

Greedy solution: <1, 2, 0, 3>

Greedy solution is

- optimal for any amount and "normal'' set of denominations
- may not be optimal for arbitrary coin denominations

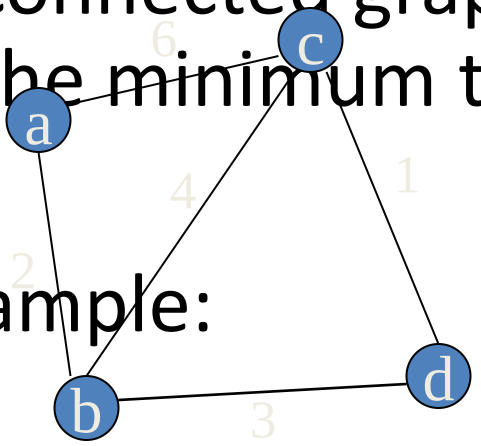
For example, $d1 = 25c$, $d2 = 10c$, $d3 = 1c$, and $n = 30c$

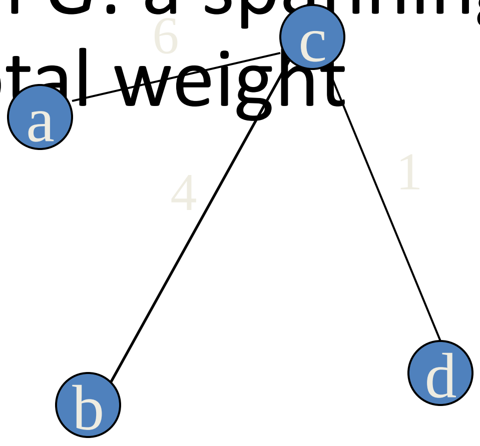
Minimum Spanning Tree (MST)

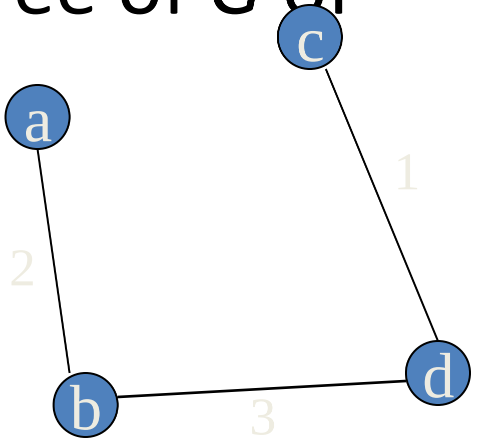
- *Spanning tree* of a connected graph *G*: a connected acyclic subgraph of *G* that includes all of *G*'s vertices

- *Minimum spanning tree* of a weighted, connected graph *G*: a spanning tree of *G* of the minimum total weight

Example:







Prim’s MST algorithm

- Start with tree T_1 consisting of one (any) vertex and “grow” tree one vertex at a time to produce MST through a series of expanding subtrees $T_1, T_2, \ldots, T_n$

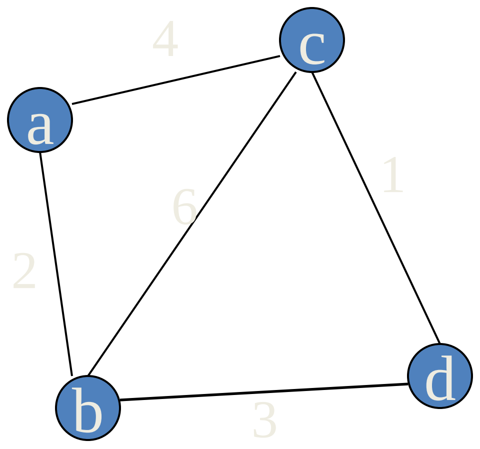
- On each iteration, construct T_{i+1} from T_i by adding vertex not in T_i that is closest to those already in T_i (this is a “greedy” step!)

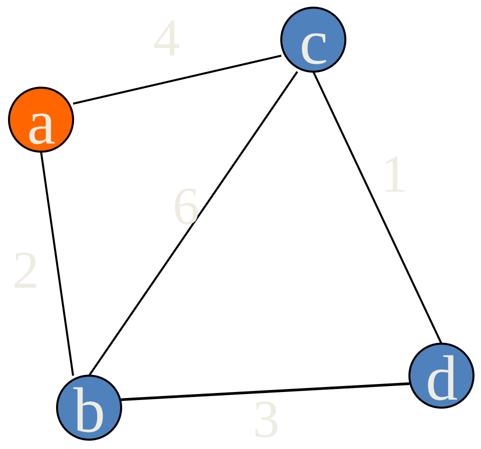
- Stop when all vertices are included

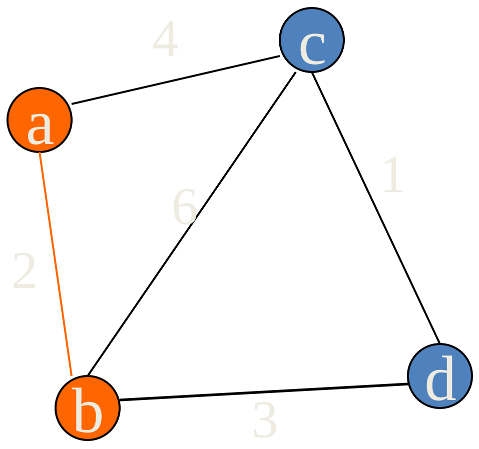
Pseudocode – Prim's algorithm

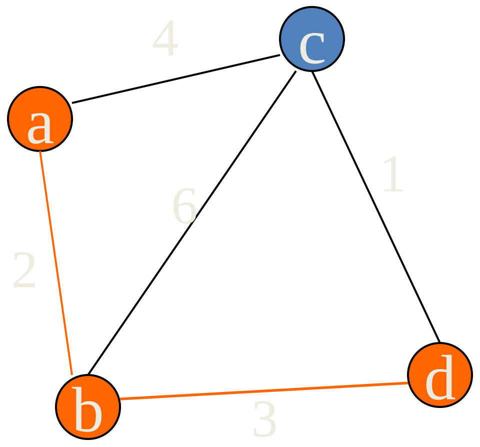
```
ALGORITHM Prim(G)
    // Prim's algorithm for computing a MST
    // Input:A weighted connected graph G = (V,E)
    // Output: Et, the set of edges composing a MST of G
    $V_T \leftarrow \{v_0\}$
    $E_T \leftarrow Ø$
    for I $\leftarrow$ 1 to |v|   - 1 do
      find a minimum weight edge e*=(v*,u*) among all edges(v,u) such
    that v is in $V_T$ and u is in $V – V_T$

      $V_T \leftarrow V_T \cup \{u^*\}$
      $E_T \leftarrow E_T \cup \{v^*\}$
    return $E_T$
```

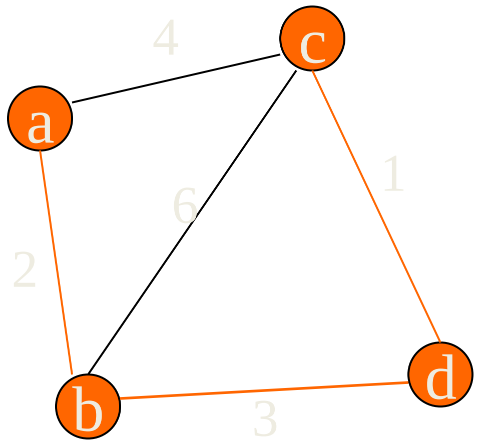
Example











Notes about Prim's algorithm

- Needs priority queue for locating closest fringe vertex.

- Efficiency
  - O(n^2) for weight matrix representation of graph and array implementation of priority queue
  - O(m log n) for adjacency lists representation of graph with n vertices and m edges and min-heap implementation of the priority queue

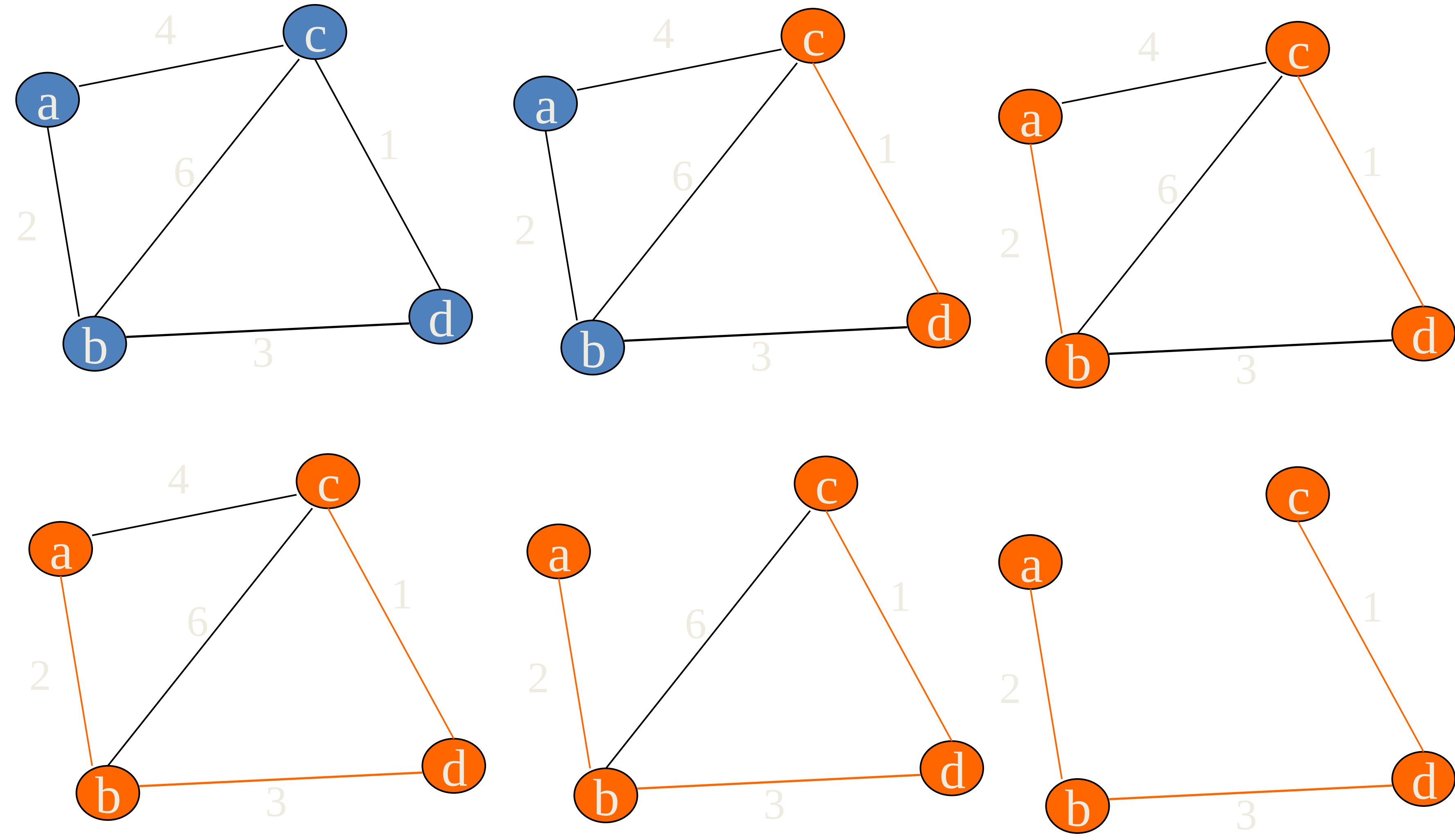
Another greedy algorithm for MST: Kruskal's

- Sort the edges in nondecreasing order of lengths
- "Grow" tree one edge at a time to produce MST through a series of expanding forests F_1, F_2, …, F_{n-1}
- On each iteration, add the next edge on the sorted list unless this would create a cycle. (If it would, skip the edge.)

Pseudocode – Kruskal’s algorithm

```
ALGORITHM Kruskal(G)
// Kruskal’s algorithm for constructing a minimum spanning tree
// Input: A weighted connected graph G = (V,E)
// Output: $E_{T,}$ The set of edges composing a MST of G
    $E_T \leftarrow \emptyset$; ecounter $\leftarrow 0$
    k $\leftarrow 0$
    while ecounter < |V| - 1
        k $\leftarrow$ k +1
        if $E_T \cup \{e_{ik}\}$ is acyclic
            $E_T \leftarrow E_T \cup \{e_{ik}\}$ ;
            ecounter $\leftarrow$ ecounter + 1
    return $E_T$
```

Example



Notes about Kruskal’s algorithm

- Algorithm looks easier than Prim’s but is harder to implement (checking for cycles!)
- Cycle checking: a cycle is created iff added edge connects vertices in the same connected component
- Kruskal’s *algorithm* relies on a *union-find algorithm* for checking cycles
- Runs in *O(m log m)* time, with *m = |E|*. The time is mostly spent on sorting.

Disjoint Sets

- Union of two sets A and B, denoted by A $\bigcup$ B, is the set {x | x $\in$ A or x $\in$ B}
- The intersection of two sets A and B, denoted by A ∩ B, is the set {x| x $\in$ A and x $\in$ B}.
- Two sets A and B are said to be disjoint if A ∩ B = ϕ.
- If *S* = {1,2,…,11} and there are 4 subsets {1,7,10,11} , {2,3,5,6}, {4,8} and {9}, these subsets may be labeled as 1, 3, 8 and 9, in this order.

Disjoint Sets

- A disjoint-set is a collection ©={S_1, S_2,…, S_k} of distinct dynamic sets.

- Each set is identified by a member of the set, called *representative*.

- Disjoint-set data structures can be used to solve the union-find problem

Disjoint set operations:

  - MAKE-SET(*x*): create a new set with only *x*. assume *x* is not already in some other set.
  - UNION(*x*,*y*): combine the two sets containing *x* and *y* into one new set. A new representative is selected.
  - FIND-SET(*x*): return the representative of the set containing *x*.

The Union-Find problem

- *N* balls initially, each ball in its own bag
  - Label the balls 1, 2, 3, ..., *N*
- Two kinds of operations:
  - Pick two bags, put all balls in these bags into a new bag (Union)
  - Given a ball, find the bag containing it (Find)

OBJECTIVE

- to design efficient algorithms for Union & Find operations.
- Approach: to represent each set as a rooted tree with data elements stored in its nodes.
- Each element *x* other than the root has a pointer to its parent *p*(*x*) in the tree.
- The root has a *null* pointer, and it serves as the name or set representative of the set.
- This results in a forest in which each tree corresponds to one set.
- For any element *x*, let *root*(*x*) denote the root of the tree containing *x*.
  - FIND(*x*) returns *root*(*x*).
  - union(*x, y*) means UNION(*root*(*x*)*, root*(*y*)).

Implementation of FIND and UNION

- FIND(*x*) → follow the path from *x* until the root is reached, then return *root*(*x*).
  - Time complexity is O(n)
  - Find(x) = Find(y), when x and y are in the same set

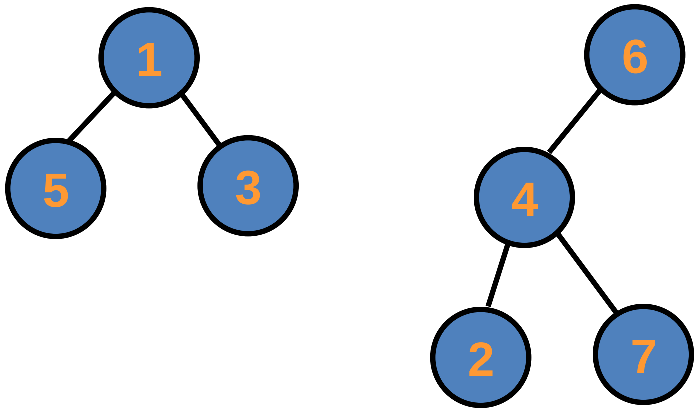
- UNION(x,y) → UNION(FIND(*x*) , FIND(*y*)) → UNION(root(*x*) , root(*y*)) → UNION(u,v) then let *v* be the parent of *u*. Assume u is root(x), v is root(y)
  - Time complexity is O(n)
  - Union(x, y) Combine the set that contains x with the set that contains y

The Union-Find problem

- An example with 4 balls
- Initial: {1}, {2}, {3}, {4}
- Union {1}, {3} $\rightarrow$ {1, 3}, {2}, {4}
- Find 3. Answer: {1, 3}
- Union {4}, {1,3} $\rightarrow$ {1, 3, 4}, {2}
- Find 2. Answer: {2}
- Find 1. Answer {1, 3, 4}

Forest Representation

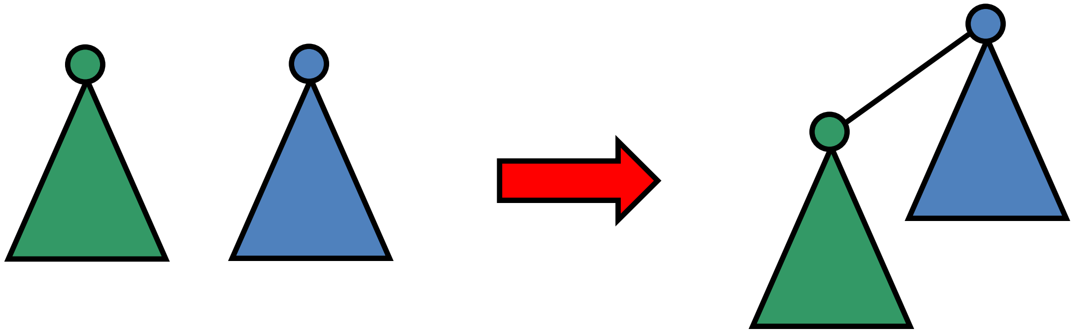
- A forest is a collection of trees
- Each bag is represented by a rooted tree, with the root being the representative ball



Example: Two bags --- {1, 3, 5} and {2, 4, 6, 7}.

Forest Representation

- Find(*x*)
  - Traverse from *x* up to the root
- Union(*x*, *y*)
  - Merge the two trees containing *x* and *y*



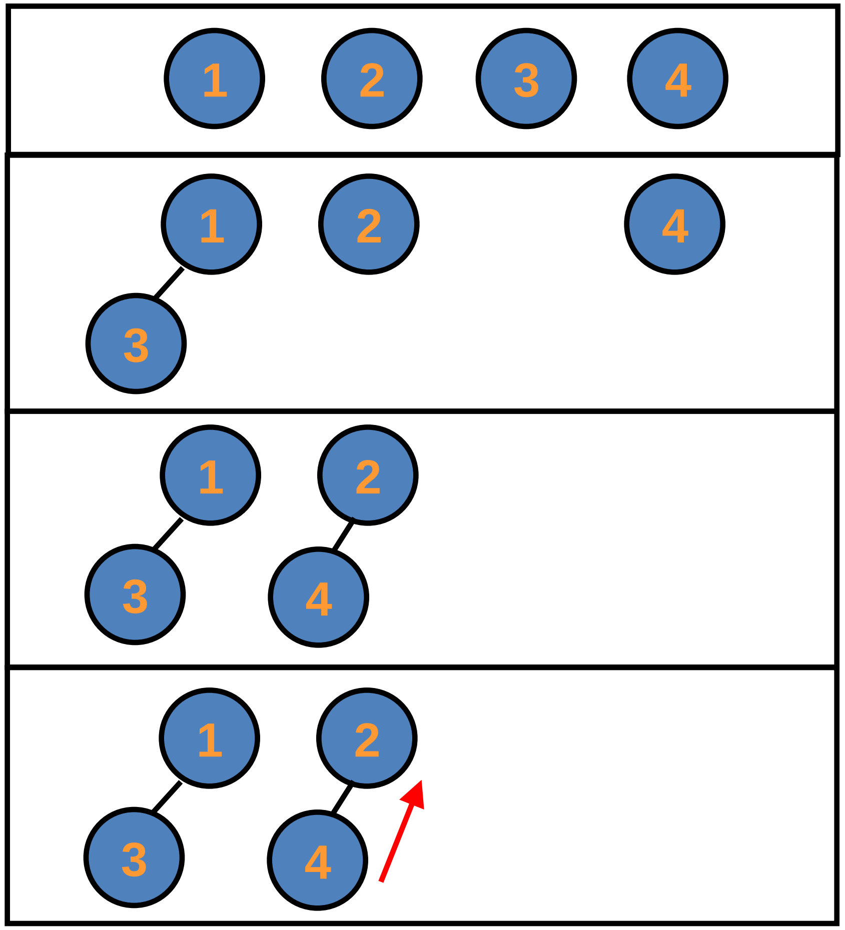
Forest Representation

Initial:

Union 1 3:

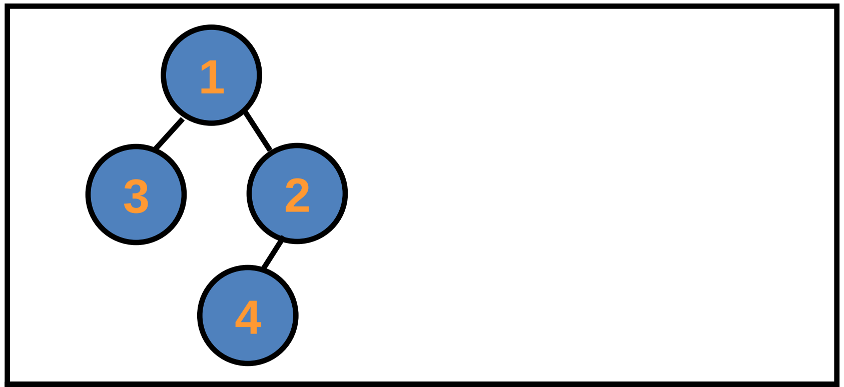
Union 2 4:

Find 4:

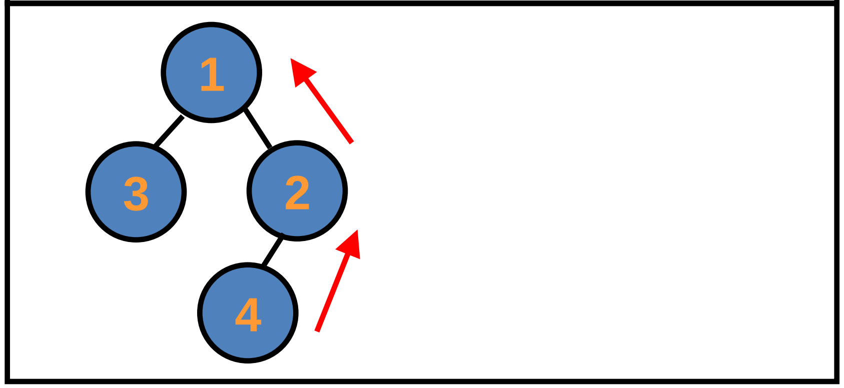


Forest Representation

Union 1 4:



Find 4:

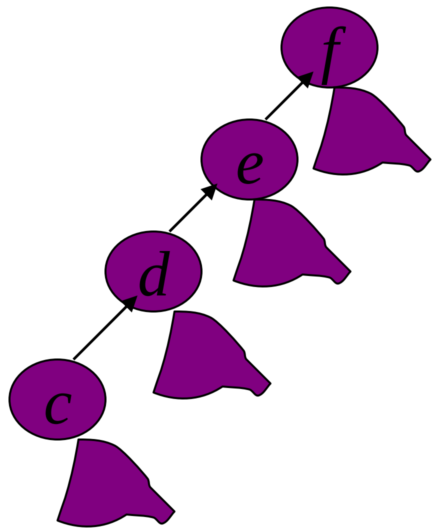


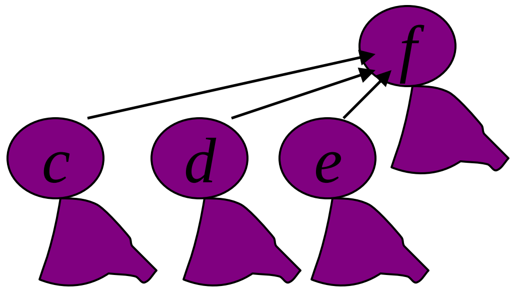
Union by Rank & Path Compression

- Union by Rank: Each node is associated with a rank, which is the upper bound on the height of the node (i.e., the height of subtree rooted at the node), then when UNION, let the root with smaller rank point to the root with larger rank.

- Path Compression: used in FIND-SET(*x*) operation, make each node in the path from *x* to the root directly point to the root. Thus reduce the tree height.

Path Compression





Shortest paths – Dijkstra's algorithm

Single Source Shortest Paths Problem: Given a weighted connected (directed) graph G, find shortest paths from source vertex *s* to each of the other vertices

Dijkstra's algorithm: Similar to Prim's MST algorithm, with a different way of computing numerical labels: Among vertices not already in the tree, it finds vertex *u* with the smallest sum

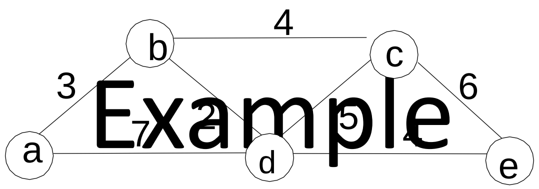
$$d_v + w(v,u)$$

where

- v is a vertex for which shortest path has been already found on preceding iterations (such vertices form a tree rooted at s)
- d_v is the length of the shortest path from source s to v
- $w(v,u)$ is the length (weight) of edge from v to u

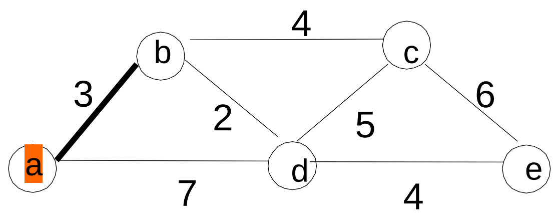
Pseudocode – Dijkstra's algorithm

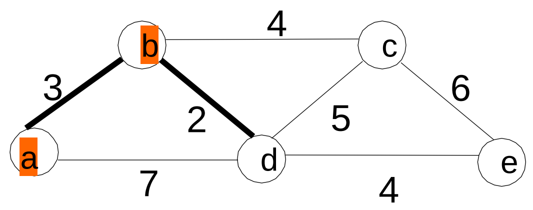
```
ALGORITHM Dijkstra(G,S)
  // Dijkstra's algorithm for single source shortest paths
  //  Input: A weighted connected graph G= (V,E) and its vertex s
  // Output: The length $d_v$ of a shortest path from s to v and its
               penultimate vertex $p_v$ for every vertex v in V
    Initialize(Q)
    for every vertex v in V do
        $d_v \leftarrow \infty$;$p_v$=null
        Insert(Q,v,$d_v$)
    $d_s \leftarrow 0$; Decrease(Q, s, $d_s$)
    $V_T \leftarrow \emptyset$
    for I $\leftarrow$ 1 to |v|   - 1 do
       u* $\leftarrow$ DeleteMin(Q)
       $V_T \leftarrow V_T \cup \{u^*\}$
       for every vertex u in V - $V_T$ that is adjacent to u* do
             if $d_u + w(u^*, u) < d_u$
                 $d_u \leftarrow d_{u^*} + w(u^*,u)$;          $p_u \leftarrow u^*$
                 Decrease (Q,u, $d_u$ )
```

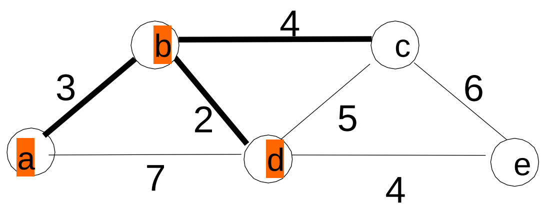


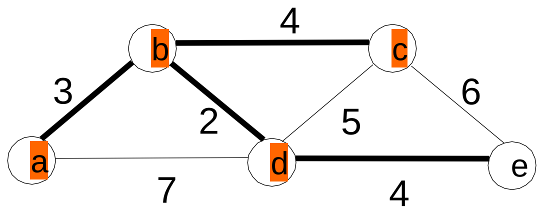
Example

Tree vertices	Remaining vertices
a(-,0)	b(a,3) c(-,∞) d(a,7) e(-,∞)
b(a,3)	c(b,3+4) d(b,3+2) e(-,∞)
d(b,5)	c(b,7) e(d,5+4)
c(b,7)	e(d,9)
e(d,9)	









Notes on Dijkstra’s algorithm

- Doesn’t work for graphs with negative weights (whereas Floyd’s algorithm does, as long as there is no negative cycle).
- Applicable to both undirected and directed graphs
- Efficiency
  - $O(|V|^2)$ for graphs represented by weight matrix and array implementation of priority queue
  - $O(|E|\log|V|)$ for graphs represented by adj. lists and min-heap implementation of priority queue

Graphs

Minimum Spanning Tree

PLSD210

Key Points

- Dynamic Algorithms
- Optimal Binary Search Tree
  - Used when
    - some items are requested more often than others
    - frequency for each item is known
  - Minimises cost of *all* searches
  - Build the search tree by
    - Considering all trees of size 2, then 3, 4, ...
    - Larger tree costs computed from smaller tree costs
      - Sub-trees of optimal trees are optimal trees!
    - Construct optimal search tree by saving root of each optimal sub-tree and tracing back
    - $O(n^3)$ time / $O(n^2)$ space

Key Points

- Other Problems using Dynamic Algorithms
- Matrix chain multiplication
  - Find optimal parenthesisation of a matrix product
    - Expressions within parentheses
      - optimal parenthesisations themselves
    - Optimal sub-structure characteristic of dynamic algorithms
    - Similar to optimal binary search tree
- Longest common subsequence
  - Longest string of symbols found in each of two sequences
- Optimal triangulation
  - Least cost division of a polygon into triangles
  - *Maps to matrix chain multiplication*

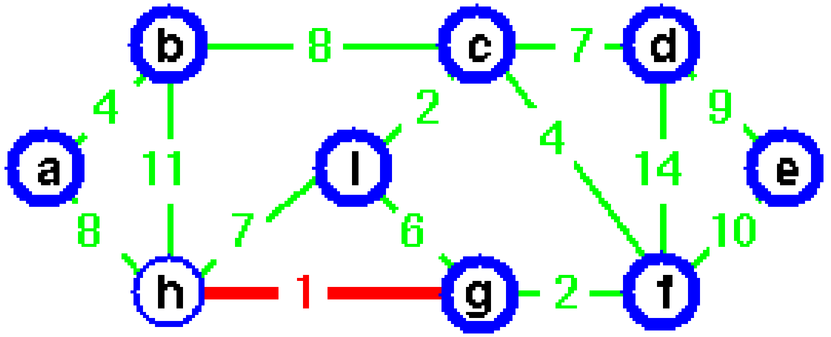
Graphs - Definitions

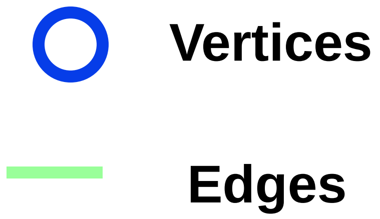
- Graph
  - Set of vertices (nodes) and edges connecting them
  - Write

$$G = (V, E)$$

where

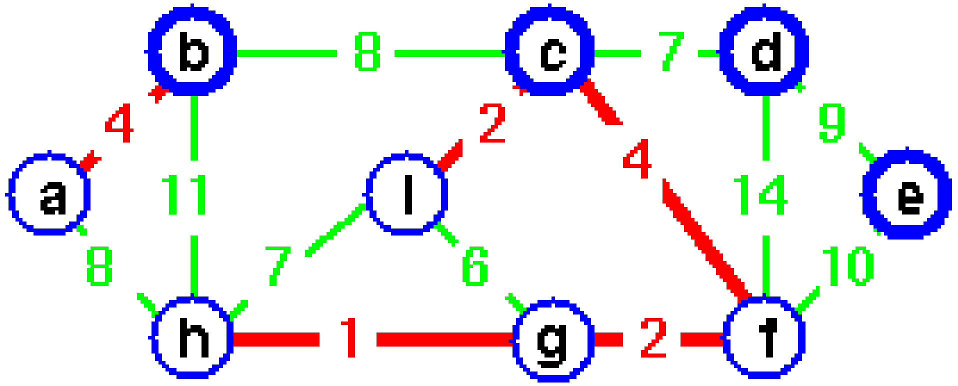
- V is a set of vertices: $V = \{ v_i \}$
- An edge connects two vertices: $e = (v_i , v_j)$
- E is a set of edges: $E = \{ (v_i , v_j) \}$





Graphs - Definitions

- Path
  - A path, p, of length, k, is a sequence of connected vertices
  - $p = \langle v, v, \ldots v \rangle$ *where* $(v_i, v_{i+1}) \in E$



< i, c, f, g, h >

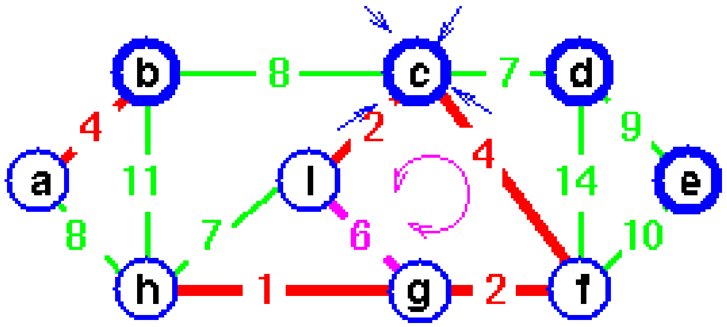
Path of length 5

< a, b >

Path of length 2

Graphs - Definitions

- Cycle
  - A graph contains no cycles if there is *no* path
  - $p = <v_0, v_1, \ldots, v_k>$ *such that* $v_0 = v_k$

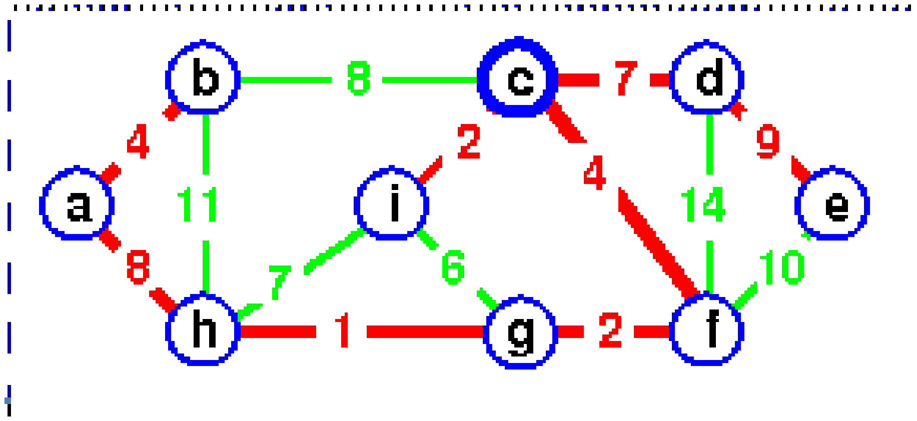


< i, c, f, g, i >

is a cycle

Graphs - Definitions

- Spanning Tree
  - A spanning tree is a set of $|V|$-1 edges that connect all the vertices of a graph



The red path connects all vertices, so it's a spanning tree

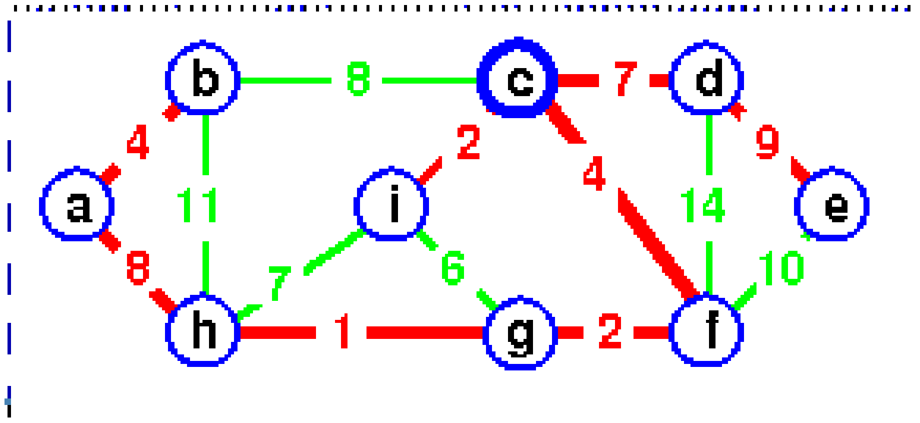
Graphs - Definitions

- Minimum Spanning Tree
  - Generally there is more than one spanning tree
  - If a cost c_{ij} is associated with edge $e_{ij} = (v_i, v_j)$

    then the minimum spanning tree is the set of edges E_{span} such that

    $$C = \Sigma \,(\, c_{ij} \mid \forall\, e_{ij} \in Espan \,)$$

    is a minimum



Other ST's can be formed ..
- **Replace 2 with 7**
- **Replace 4 with 11**

The red tree is the Min ST

Graphs - Kruskal's Algorithm

- Calculate the minimum spanning tree
  - Put all the vertices into single node trees by themselves
  - Put all the edges in a priority queue
  - Repeat until we've constructed a spanning tree
    - *Extract cheapest edge*
    - If it forms a cycle, ignore it
      else add it to the forest of trees
      (it will join two trees into a larger tree)
  - Return the spanning tree

Graphs - Kruskal's Algorithm

- Calculate the minimum spanning tree
  - Put all the vertices into single node trees by themselves
  - Put all the edges in a priority queue
  - Repeat until we've constructed a spanning tree
    - *Extract cheapest edge*
    - If it forms a cycle, ignore it
      else add it to the forest of trees
      (it will join two trees into a larger tree)
  - Return the spanning tree

Note that this algorithm makes no attempt
• to be clever
• to make any sophisticated choice of the next edge
• it just tries the cheapest one!

Graphs - Kruskal’s Algorithm in C

```
Forest MinimumSpanningTree( Graph g, int n,
                            double **costs ) {
    Forest T;
    Queue q;
    E
    T = Co
    q = ConsEdgeQueue( g,
    for(i=0;i<(n-1);i+
        do {
            e = ExtractCheapestEdge( q );
        } while ( !Cycle( e, T ) );
        AddEdge( T, e );
    }
    return T;
}
```

Initial Forest: single vertex trees

P Queue of edges

Graphs - Kruskal's Algorithm in C

```
Forest MinimumSpanningTree( Graph g, int n,
                            double **costs ) {
    Forest T;
    T = Co
    q = ConsEdgeQueue( g, costs );
    for(i=0;i<(n-1);i++) {
        do {
            e = ExtractCheapestEdge( q );
        } while ( !Cycle( e, T ) );
        AddEdge( T, e );
    }
    return T;
}
```

We need n-1 edges to fully connect (span) n vertices

Graphs - Kruskal’s Algorithm in C

```
Forest MinimumSpanningTree( Graph g, int n,
                            double **costs ) {
    Forest T;
    Queue q;
    Edge e;
    T = ConsForest( g );
    q = ConsEdgeQueue( g, costs );
    for(i=0
        do {
            e = ExtractCheapestEdge( q );
        } while ( !Cycle( e, T ) );
        AddEdge(
    }
```

Try the cheapest edge

Until we find one that doesn’t form a cycle

... and add it to the forest

Kruskal’s Algorithm

- Priority Queue
  - We already know about this!!

```
Forest MinimumSpanningTree( Graph g, int n,
                            double **costs ) {
    Forest T;
    Queue q;
    Edge e;
    T = ConsForest( g );
    q = ConsEdgeQueue( g, costs );
    for(i=0;i<(n-1);i++) {
        do {
            e = ExtractCheapestEdge( q );
        } while ( !Cycle( e, T ) );
        AddEdge( T, e );
    }
    return T;
}
```

Add to a heap here

Extract from a heap here

Kruskal’s Algorithm

- Cycle detection

```
Forest MinimumSpanningTree( Graph g, int n,
                            double **costs ) {
    Forest T;
    Queue q;
    Edge e;
    T = ConsForest( g );
    q = ConsEdgeQueue( g, costs );
    for(i=0;i<(n-1);i++) {
        do {
            e = ExtractCheapestEdge(
        } while ( !Cycle( e,
        AddEdge( T, e );
    }
    return T;
}
```

But how do we detect a cycle?

Kruskal's Algorithm

- Cycle detection
  - Uses a Union-find structure
  - For which we need to understand a partition of a set
- Partition
  - A set of sets of elements of a set
    - Every element belongs to one of the sub-sets
    - No element belongs to more than one sub-set
  - Formally:
    - Set, $S = \{ s_i \}$
    - Partition(S) $= \{ P_i \}$, where $P_i = \{ s_i \}$
    - $\forall s_i \in S, \; s_i \in P_j$

P_i are subsets of S

All s_i belong to one of the P_j

None of the P_i have common elements

S is the union of all the P_i

Kruskal’s Algorithm

- Partition
  - The elements of each set of a partition
    - are related by an equivalence relation
    - equivalence relations are
      - reflexive
      - transitive
      - symmetric

x ~ x

if x ~ y and y ~ z, then x ~ z

if x ~ y, then y ~ x

  - The sets of a partition are equivalence classes
    - Each element of the set is related to every other element

Kruskal’s Algorithm

- Partitions
  - In the MST algorithm,
    the connected vertices form equivalence classes
    - “Being connected” is the equivalence relation
  - Initially, each vertex is in a class by itself
  - As edges are added,
    more vertices become related
    and the equivalence classes grow
  - Until finally all the vertices are in a single equivalence class

Kruskal’s Algorithm

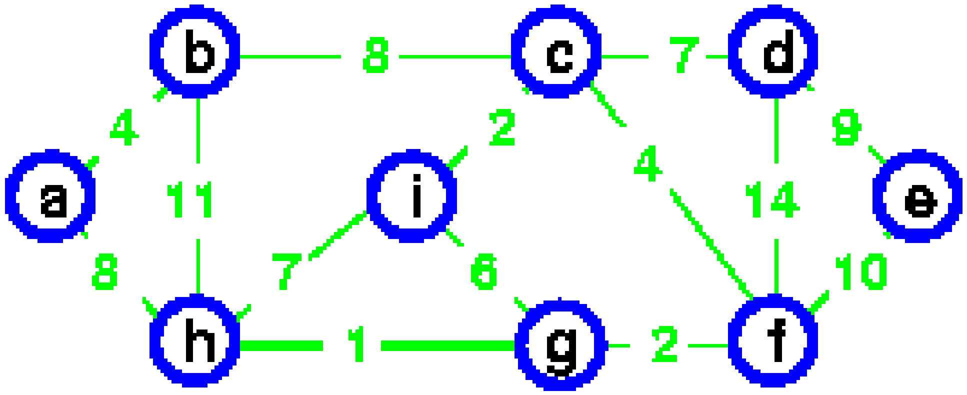
- Representatives
  - One vertex in each class may be chosen as the representative of that class
  - We arrange the vertices in lists that lead to the representative
    - This is the union-find structure
- Cycle determination

Kruskal's Algorithm

- Cycle determination
  - If two vertices have the same representative, they're already connected and adding a further connection between them is pointless
  - Procedure:
    - For each end-point of the edge that you're going to add
    - follow the lists and find its representative
    - if the two representatives are equal,
      then the edge will form a cycle

Kruskal’s Algorithm in operation

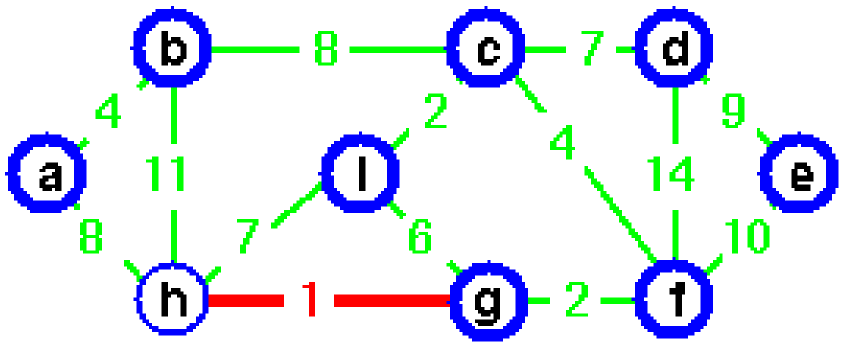
All the vertices are in single element trees



Each vertex is its own representative

Kruskal’s Algorithm in operation

All the vertices are in single element trees

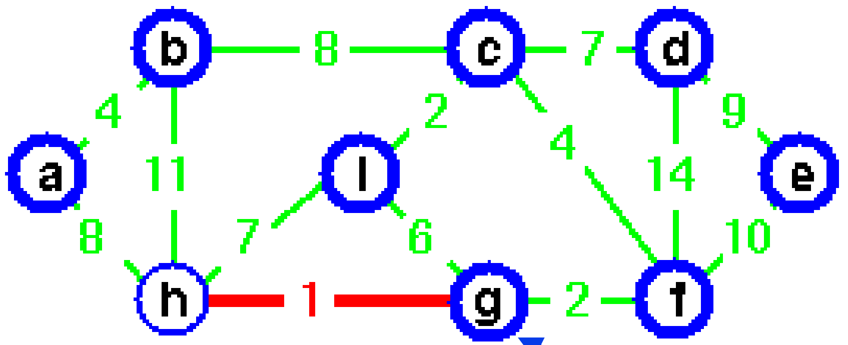


The cheapest edge is h-g

Add it to the forest, joining h and g into a 2-element tree

Kruskal's Algorithm in operation

The cheapest edge is h-g



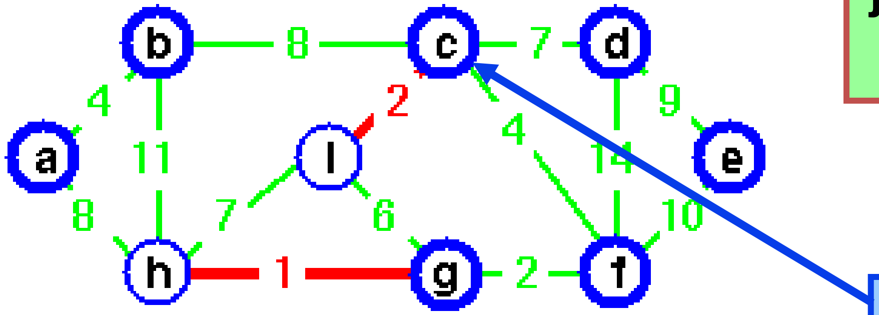
Choose g as its representative

Add it to the forest, joining h and g into a 2-element tree

Kruskal’s Algorithm in operation

The next cheapest edge is c-i

Add it to the forest, joining c and i into a 2-element tree



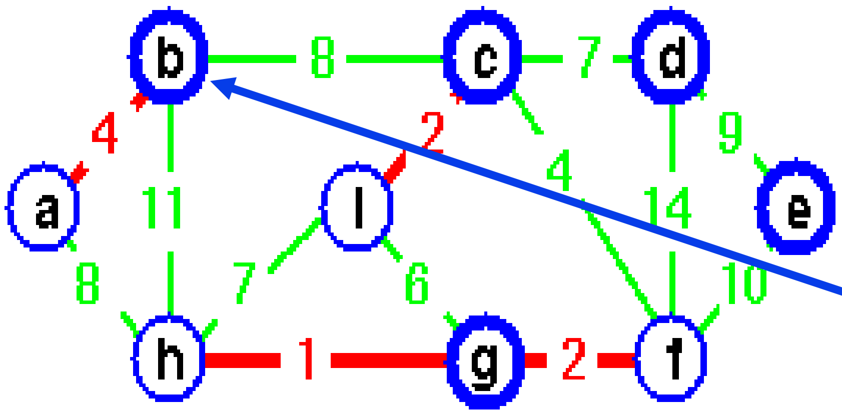
Choose c as its representative

Our forest now has 2 two-element trees and 5 single vertex ones

Kruskal’s Algorithm in operation

The next cheapest edge is a-b

Add it to the forest, joining a and b into a 2-element tree



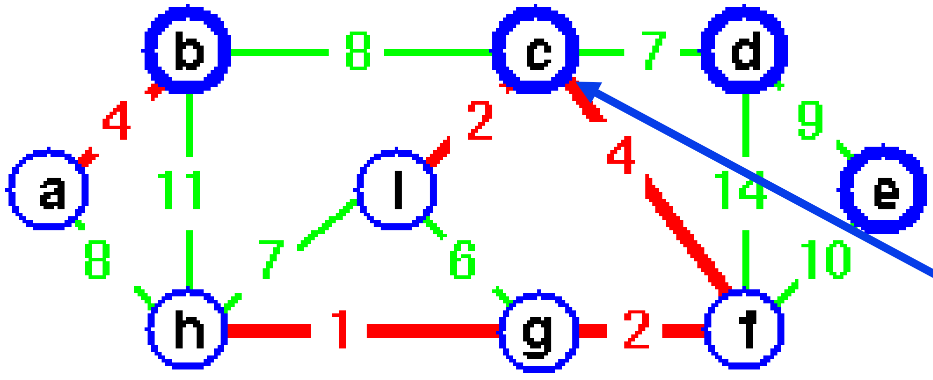
Choose b as its representative

Our forest now has 3 two-element trees and 4 single vertex ones

Kruskal's Algorithm in operation

The next cheapest edge is c-f

Add it to the forest, *merging* two 2-element trees



Choose the rep of one as its representative

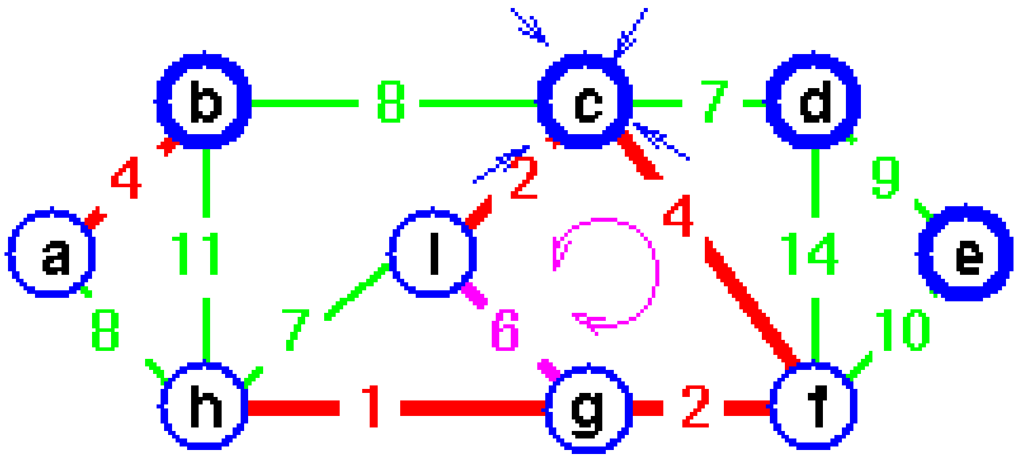
Kruskal's Algorithm in operation

The next cheapest edge is g-i

The rep of g is c

The rep of i is also c

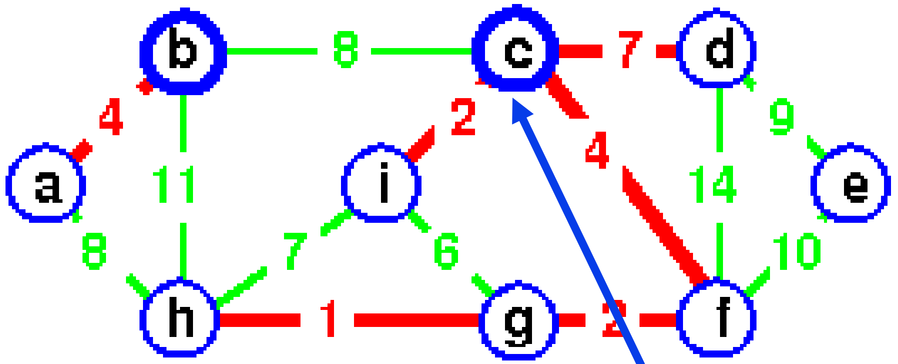
∴ g-i forms a cycle



It's clearly not needed!

Kruskal’s Algorithm in operation

The next cheapest edge is c-d



The rep of c is c

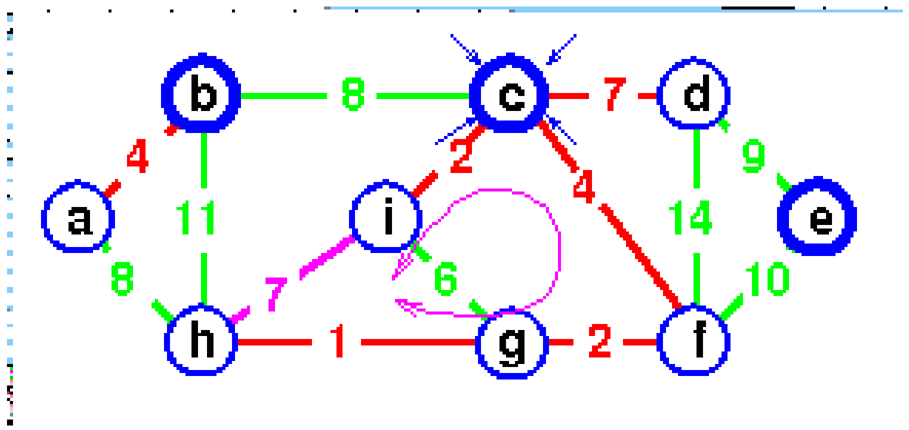
The rep of d is d

∴ c-d joins two trees, so we add it

.. and keep c as the representative

Kruskal’s Algorithm in operation

The next cheapest edge is **h-i**



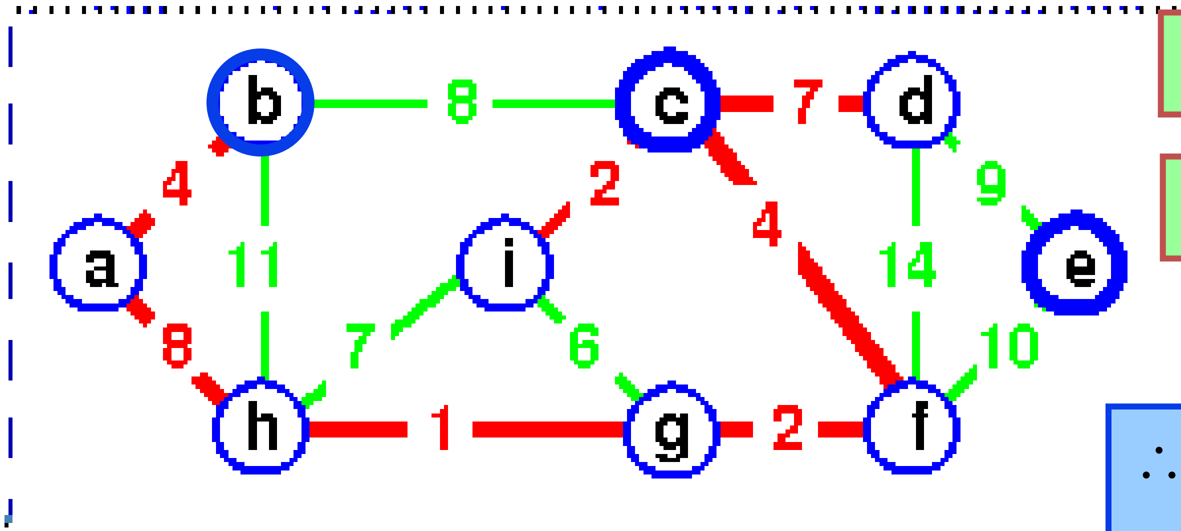
The rep of **h** is **c**

The rep of **i** is **c**

∴ **h-i** forms a cycle, so we skip it

Kruskal’s Algorithm in operation

The next cheapest edge is a-h



The rep of a is b

The rep of h is c

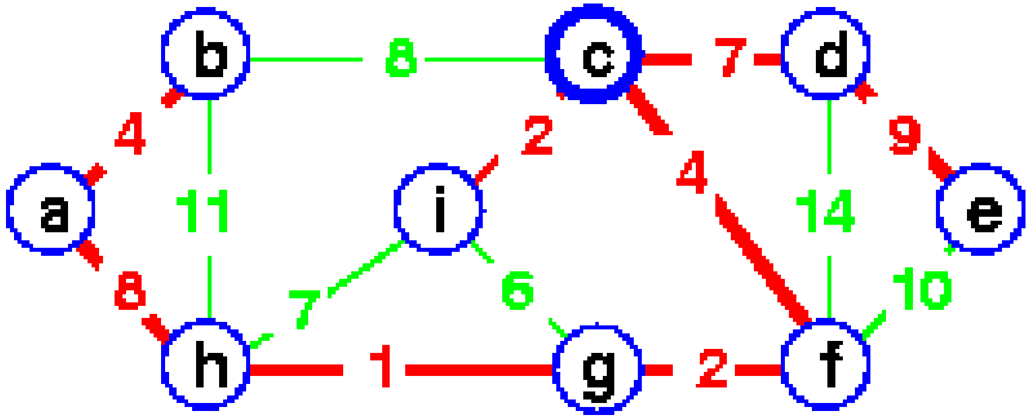
∴ a-h joins two trees, and we add it

Kruskal’s Algorithm in operation

The next cheapest edge is **b-c**

But **b-c** forms a cycle

So add **d-e** instead



... and we now have a spanning tree

Greedy Algorithms

- *At no stage did we attempt to "look ahead"*
- We simply made the naïve choice
  - Choose the cheapest edge!
- MST is an example of a greedy algorithm
- Greedy algorithms
  - Take the "best" choice at each step
  - Don't look ahead and try alternatives
  - *Don't work in many situations*
    - Try playing chess with a greedy approach!
  - *Are often difficult to prove*

Proving Greedy Algorithms

- MST Proof
  - "Proof by contradiction" is usually the best approach!
  - Note that
    - any edge creating a cycle is not needed

    ∴ Each edge must join two sub-trees
  - Suppose that the next cheapest edge, e_x, would join trees T_a and T_b
  - Suppose that instead of e_x we choose e_z - a more expensive edge, which joins T_a and T_c
  - But we still need to join T_b to T_a or some other tree

MST - Time complexity

- Steps
  - Initialise forest $O(\ |V|\ )$
  - Sort edges $O(\ |E|\log|E|\ )$
    - Check edge for cycles $O(\ |V|\ )$ x
    - Number of edges $O(\ |V|\ )$ $O(\ |V|^2\ )$
  - Total $O(\ |V|+|E|\log|E|+|V|^2\ )$
  - Since $|E| = O(\ |V|^2\ )$ $O(\ |V|^2\log|V|\ )$

  - Thus we would class MST as $O(\ n^2\log n\ )$ for a graph with n vertices

MST - Time complexity

- Steps
  - Initialise f
  - Sort edge
    - Check edge for cycles $O(\ |V|$
    - Number of edges $O(\ |V$ $O(|V|^2\ )$
  - Total $O(\ |V|+|E|\log|E|+|V|^2\ )$
  - Since $|E| = O(\ |V|^2\ )$ $O(\ |V|^2\log|V|\ \ )$

  - Thus we would class MST as $O(\ n^2\log n\ )$ for a graph with n vertices

Here's the "professionals read textbooks" theme recurring again!

UNIT-IV

BACKTRACKING

Tackling Difficult Combinatorial Problems

There are two principal approaches to tackling difficult combinatorial problems (NP-hard problems):

- ☐Use a strategy that guarantees solving the problem exactly but doesn’t guarantee to find a solution in polynomial time
- ☐Use an approximation algorithm that can find an approximate (sub-optimal) solution in polynomial time

Exact Solution Strategies

- *exhaustive search* (brute force)
  - useful only for small instances
- *dynamic programming*
  - applicable to some problems (e.g., the knapsack problem)
- *backtracking*
  - eliminates some unnecessary cases from consideration
  - yields solutions in reasonable time for many instances but worst case is still exponential
- *branch-and-bound*
  - further refines the backtracking idea for optimization problems

Backtracking

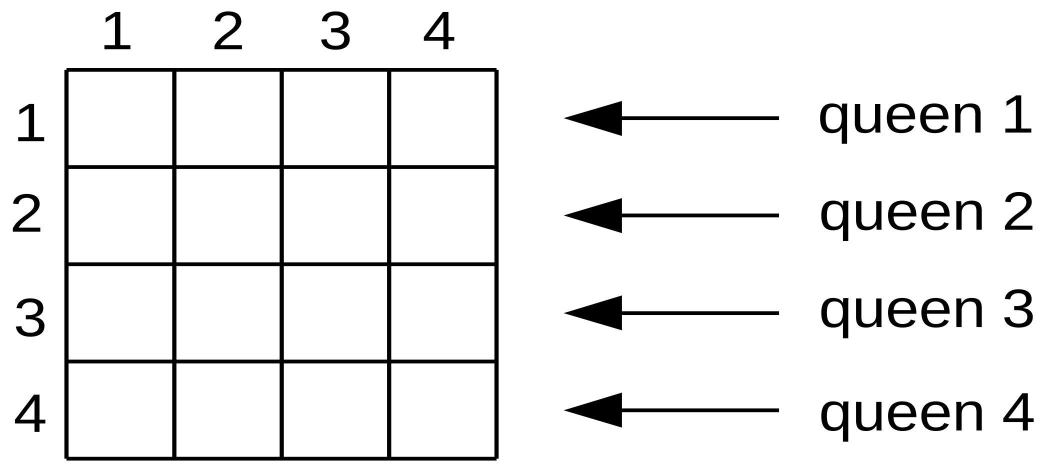
- Construct the <u>*state-space tree*</u>
  - nodes: partial solutions
  - edges: choices in extending partial solutions

- Explore the state space tree using depth-first search

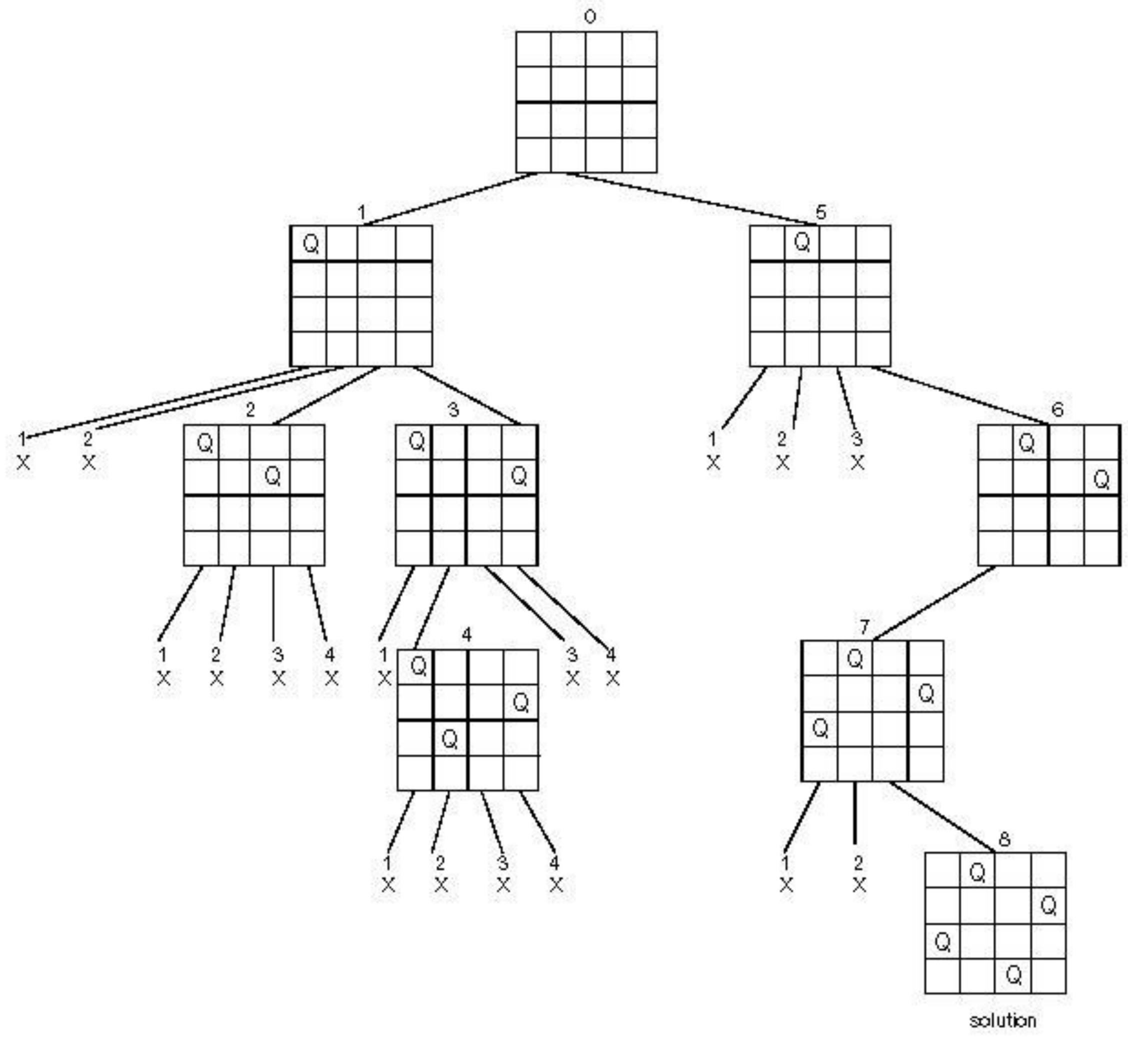
- “Prune” <u>*nonpromising nodes*</u>
  - dfs stops exploring subtrees rooted at nodes that cannot lead to a solution and backtracks to such a node’s parent to continue the search

Example: *n*-Queens Problem

Place *n* queens on an *n*-by-*n* chess board so that no two of them are in the same row, column, or diagonal



State-Space Tree of the 4-Queens Problem



n-Queens Problem

```
Algorithm NQueens(k,n)
{
  for i=1 to n do
  {
        if place(k,i) then
        {
            x[k]=i;
            if (k = n) then write (x[1:n]);
            else NQueens(k+1, n);
        }
  }
  NQueens(k-1,n);
  i= x[k-1]+1;
}
```

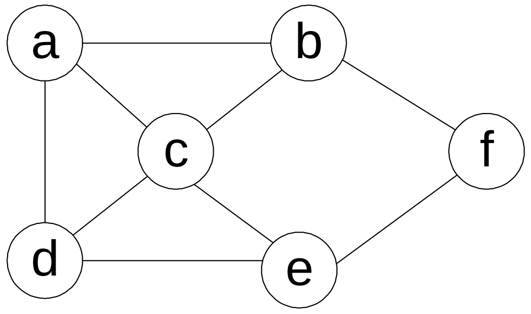
n-Queens Problem

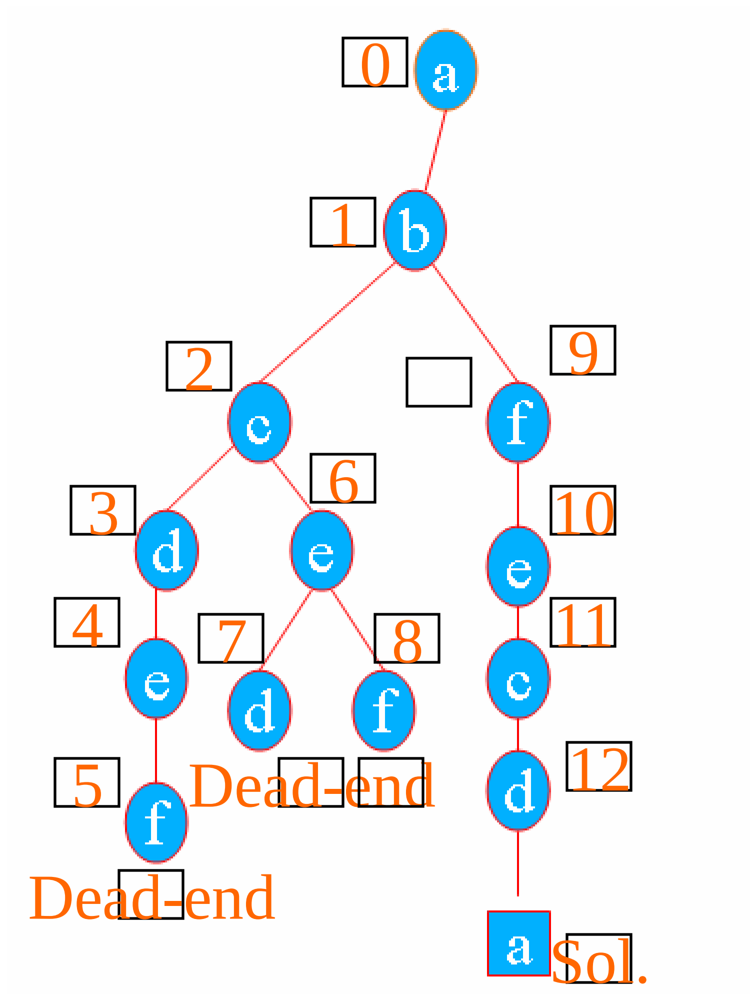
```
Algorithm place(k,i)
// To place a new queen in the chessboard
// Returns true if a queen can be placed in kth row and ith column, otherwise
   false.
// x[ ] is a global array whose first (k-1) values have been set
// Abs(r) returns the absolute value of r
{
  for j = 1 to k-1 do
    if ((x[j] = i) or (Abs (j-k))) then
       return false;
 return true;
}
```

Hamiltonian Circuit

- Hamiltonian Path: is a path in an undirected graph which visits each vertex exactly once
- Hamiltonian circuit: is a cycle in an undirected graph which visits each vertex exactly once and also returns to the starting vertex.
- Determining whether such paths and cycles exists is the hamiltonian path problem.
- Hamiltonian paths and cycles are named after William Rowan Hamilton

State Space Tree of Hamiltonian Circuit Problem





SUBSET-SUM PROBLEM

Let $S = \{s_1, \ldots\ldots, s_n\}$ be a set of positive integers, then we have to find a subset whose sum is equal to given positive integer ‘d’.

Sort the elements of the set in ascending order.

$s_1 <= s_2 <= s_3 \ldots\ldots\ldots <= s_n$

S={1,2,5,6,8}   d=9

2 solutions: {1,2,6} and {1,8}

Subset – Sum Problem

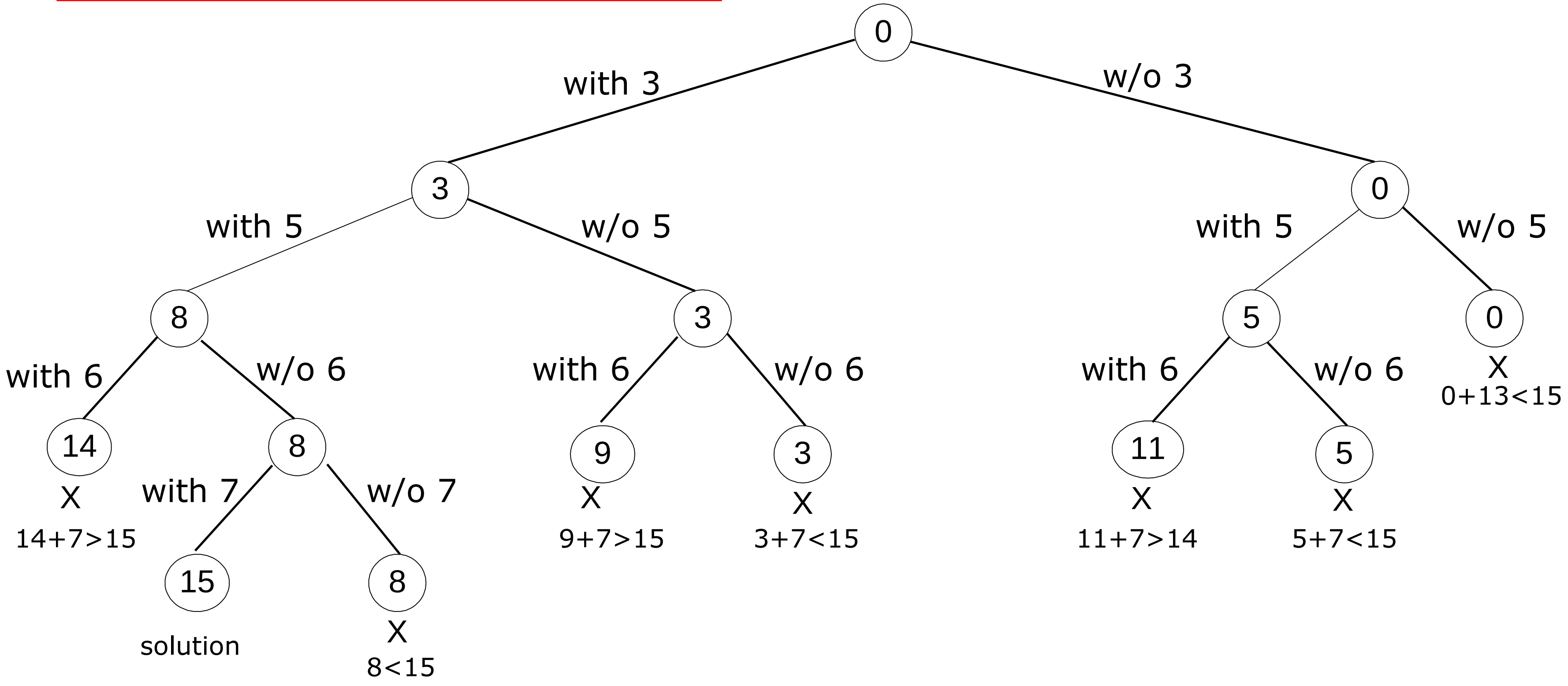
Steps:

1. The root of the tree represents the starting point, with no decisions about the given elements.
2. Its left and right children represent, inclusion and exclusion of s1 in the set being sought
3. Going to the left from a node of the first level corresponds to inclusion of s2, while going to right corresponds to exclusion
4. A path from the root to a node at the ith level of the tree indicates which of the first i numbers have been included in the subsets represented by that nodeWe record the value of s', the sum of these numbers in the node.
5. If s' =d, we have a solution to the problem and stop. If all the solutions need to be found, continue by backtracking to the node's parent.
6. If s' is not equal to d, we can terminate the node as nonpromising if either of the two equalities holds.
7. $s' + s_{i+1} > d$ (the sum s' too large)

   $s' + \sum_{j=i+1}^{n} s_j < d$ (the sum s' too small)

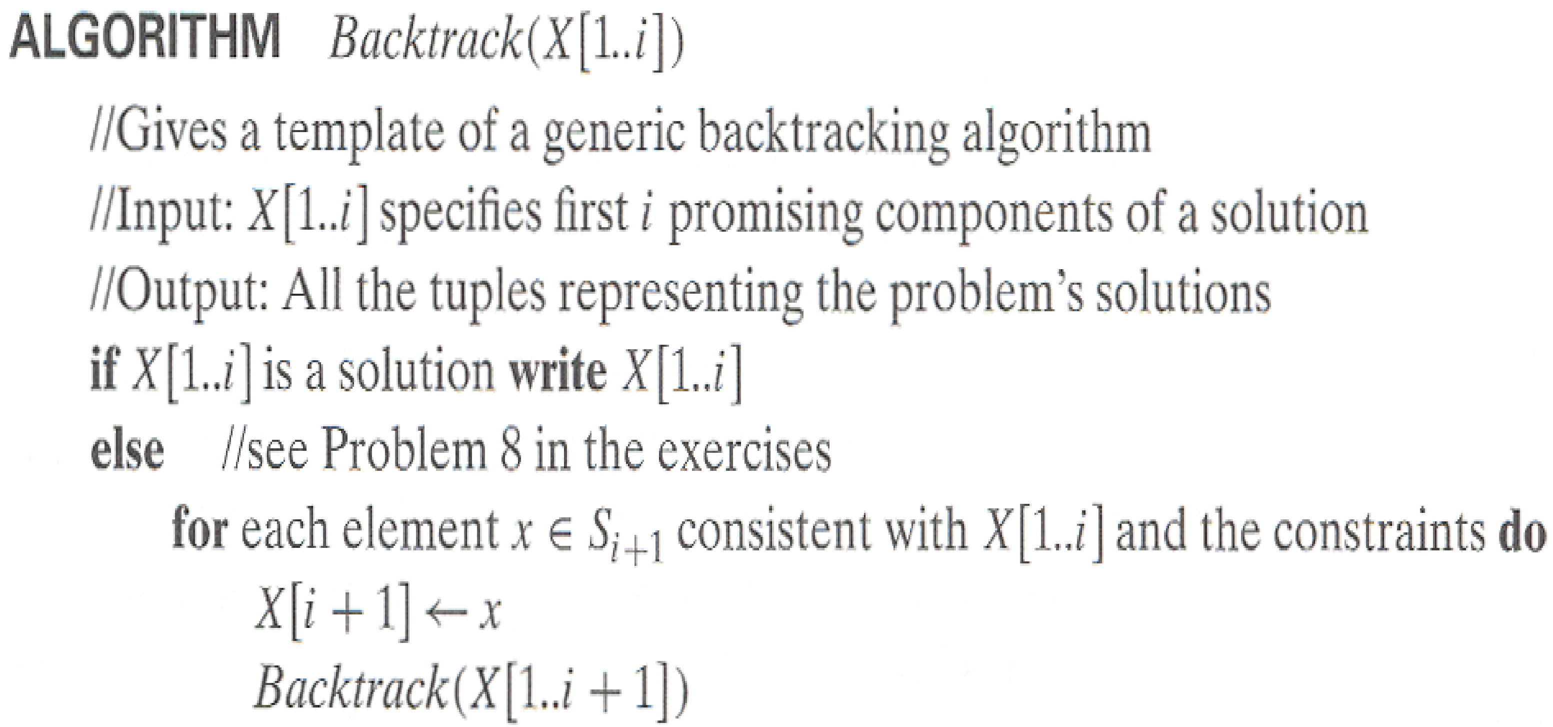
State Space Tree of Subset – Sum Problem

S= {3,5,6,7} and d=15



No. of Nodes in the state space tree is $1+2+2^2+\ldots+2^n = 2^{n+1} -1$

Pseudocode: Backtracking



```
ALGORITHM  Backtrack($X[1..i]$)
    //Gives a template of a generic backtracking algorithm
    //Input: $X[1..i]$ specifies first $i$ promising components of a solution
    //Output: All the tuples representing the problem's solutions
    if $X[1..i]$ is a solution write $X[1..i]$
    else    //see Problem 8 in the exercises
        for each element $x \in S_{i+1}$ consistent with $X[1..i]$ and the constraints do
            $X[i+1] \leftarrow x$
            Backtrack($X[1..i+1]$)
```

BRANCH AND BOUND

Branch-and-Bound

- Branch and bound (BB) is a general algorithm for finding optimal solutions of various optimization problems, especially in discrete and combinatorial optimization.

- It consists of a systematic enumeration of all candidate solutions, where large subsets of fruitless candidates are discarded, by using <u>upper and lower estimated bounds</u> of the quantity being optimized.

Branch-and-Bound

- In the standard terminology of optimization problems, a *feasible solution* is a point in the problem’s search space that satisfies all the problem’s constraints

- An *optimal solution* is a feasible solution with the best value of the objective function

Branch-and-Bound

- 3 Reasons for terminating a search path at the current node in a state-space tree of a branch-and-bound algorithm:

1. The value of the node's bound is <u>not better </u>than the value of the best solution seen so far.

2. The node represents <u>no feasible solutions </u>because the constraints of the problem are already violated.

3. The subset of feasible solutions represented by the node consists of a single point—in this case we compare the value of the objective function for this feasible solution with that of the best solution seen so far and update the latter with the former if the new solution is better.

Branch-and-Bound

- An enhancement of backtracking
- Applicable to optimization problems
- For each node (partial solution) of a state-space tree, computes a bound on the value of the objective function for all descendants of the node (extensions of the partial solution)
- Uses the bound for:
  - ruling out certain nodes as “nonpromising” to prune the tree – if a node’s bound is not better than the best solution seen so far
  - guiding the search through state-space

Example: Assignment Problem

Select one element in each row of the cost matrix *C* so that:
- **no two selected elements are in the same column**
- **the sum is minimized**

Example

	Job 1	Job 2	Job 3	Job 4
Person *a*	9	2	7	8
Person *b*	6	4	3	7
Person *c*	5	8	1	8
Person *d*	7	6	9	4

Lower bound: Any solution to this problem will have total cost at least: 2 + 3 + 1 + 4 (or 5 + 2 + 1 + 4)

Example: First two levels of the state-space tree

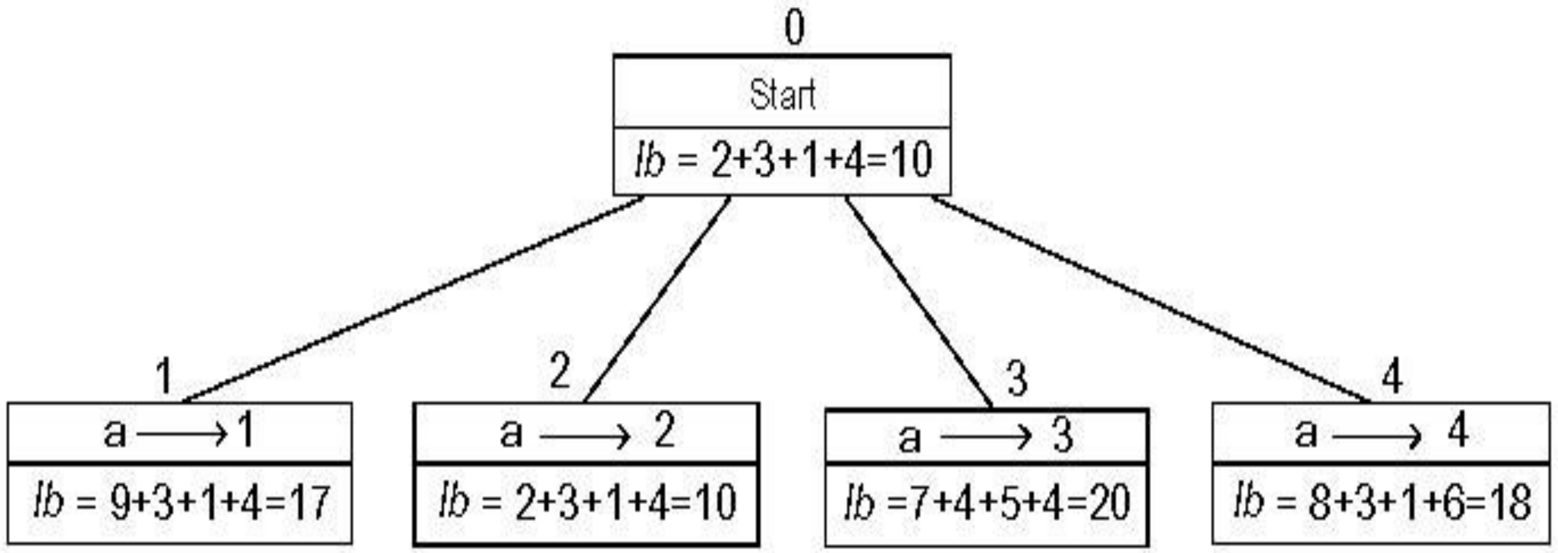


Figure 11.5 Levels 0 and 1 of the state-space tree for the instance of the assignment problem being solved with the best-first branch-and-bound algorithm. The number above a node shows the order in which the node was generated. A node's fields indicate the job number assigned to person *a* and the lower bound value, *lb*, for this node.

Example (cont.)

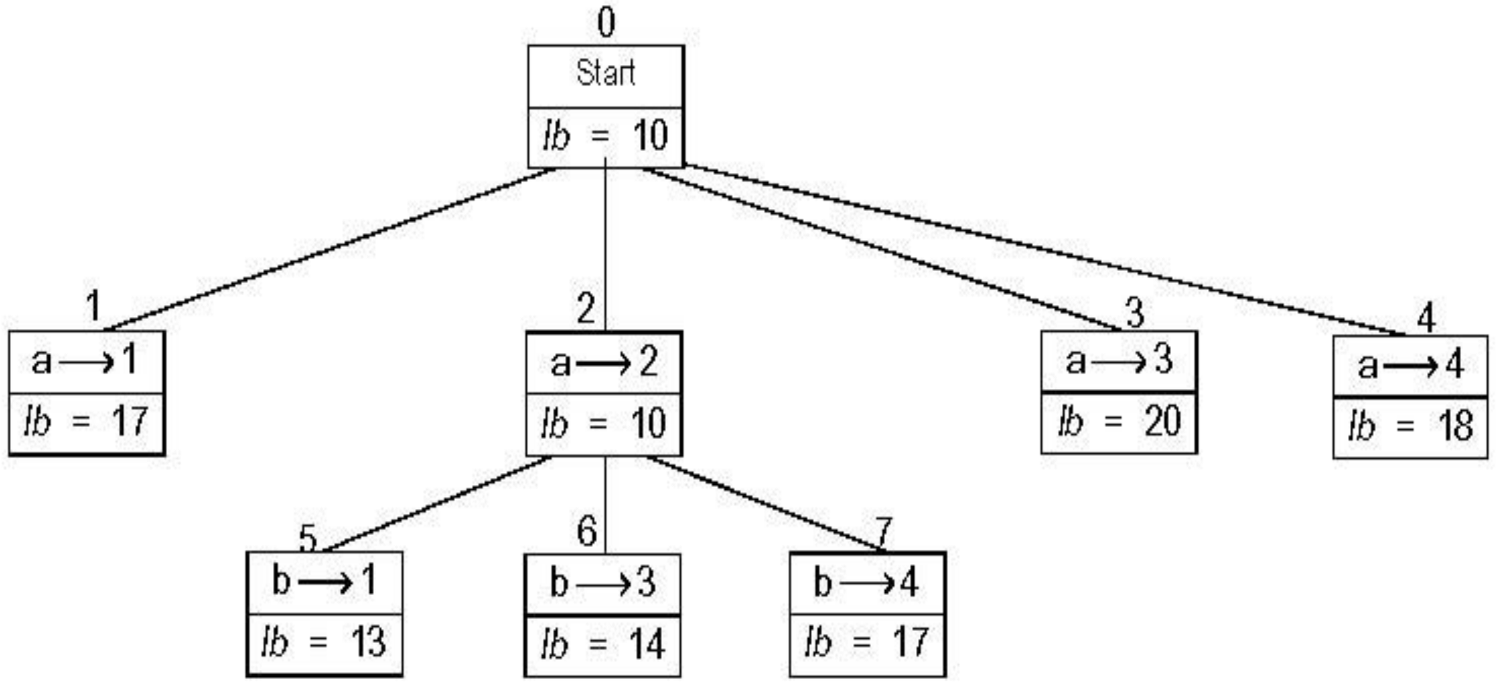


Figure 11.6 Levels 0, 1, and 2 of the state-space tree for the instance of the assignment problem being solved with the best-first branch-and-bound algorithm

Example: Complete state-space

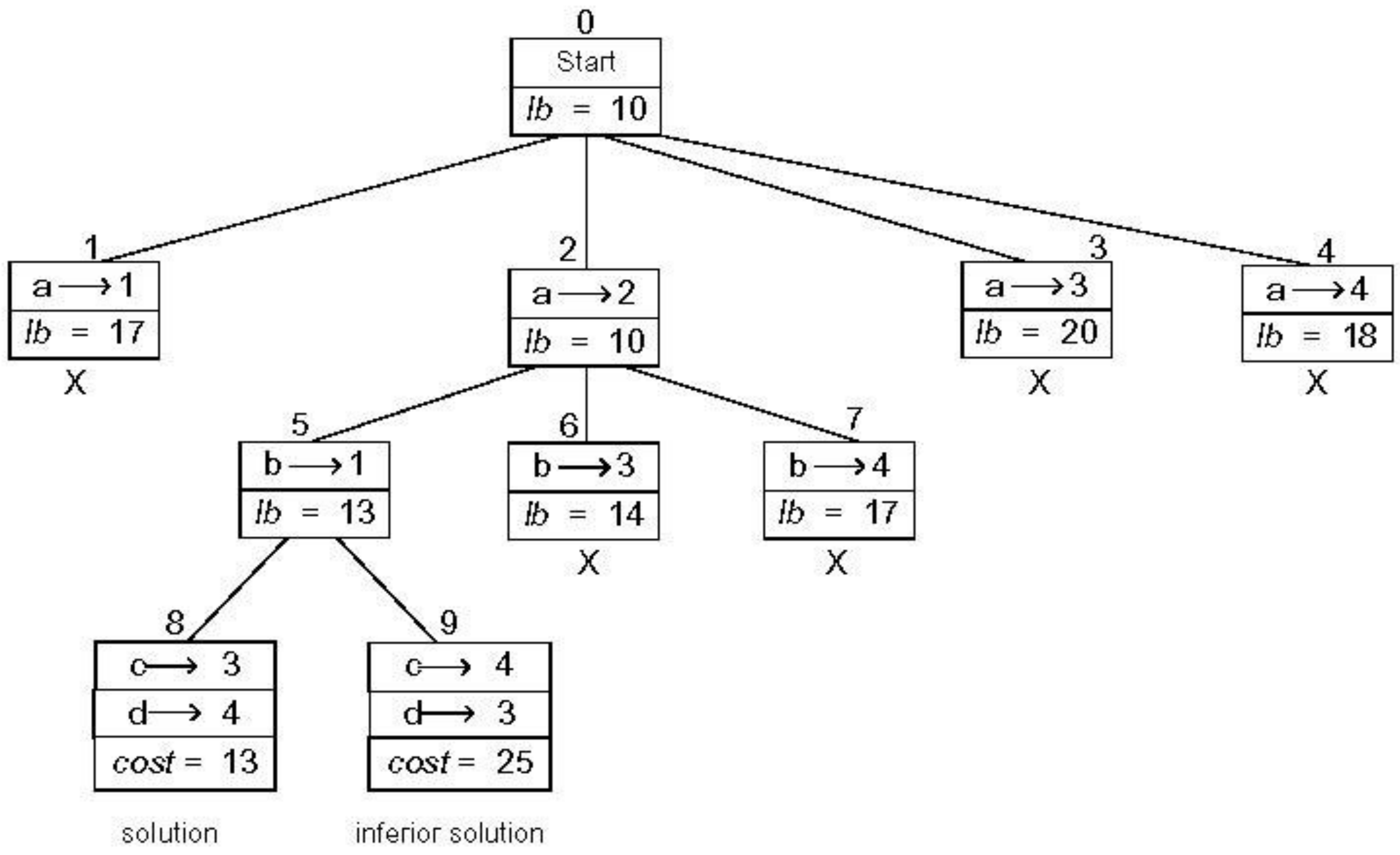


Figure 11.7 Complete state-space tree for the instance of the assignment problem solved with the best-first branch-and-bound algorithm

Solution:

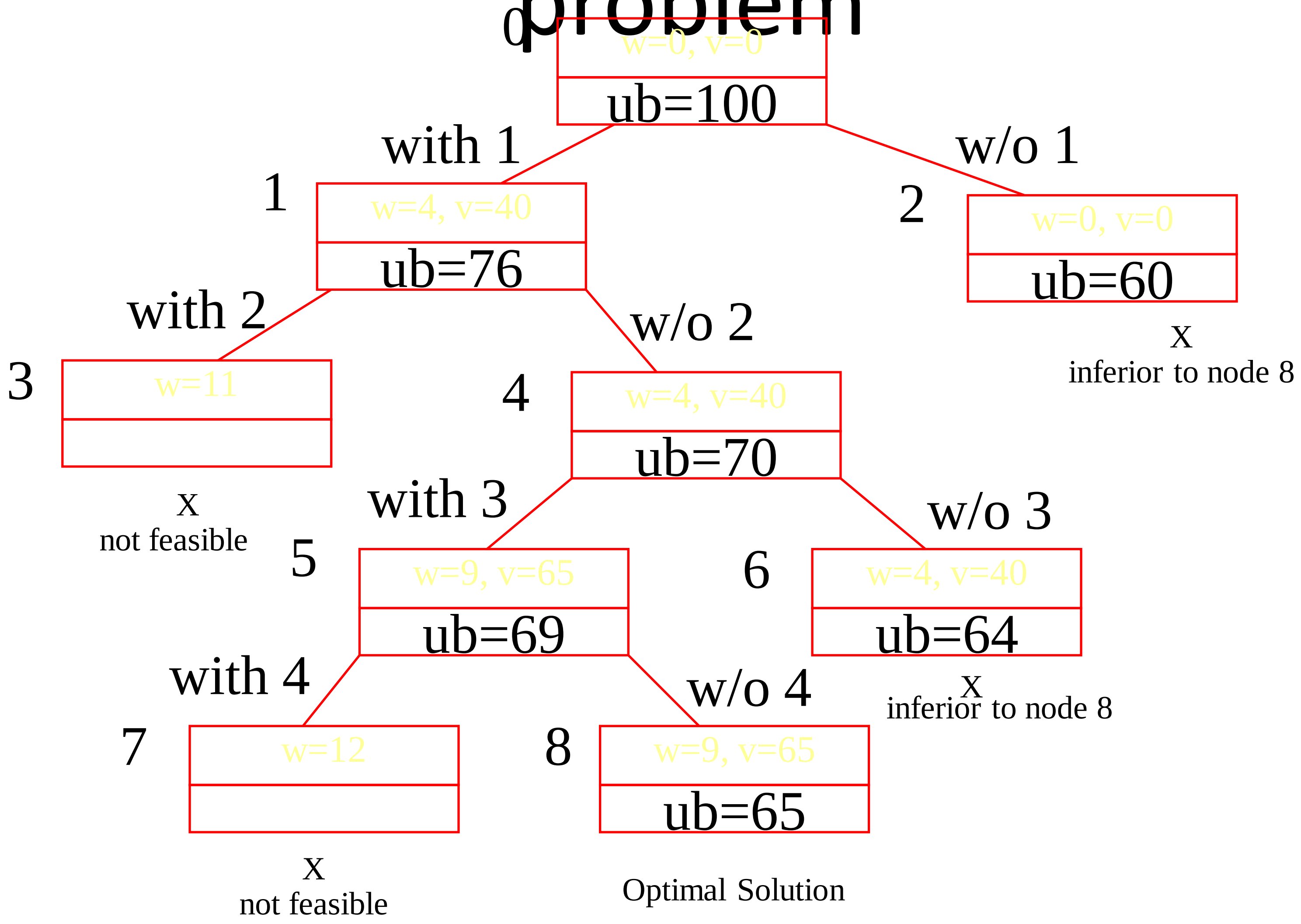
- Person a – job 2
- Person b – job 1
- Person c – job 3
- Person d – job 4

KNAPSACK PROBLEM

- N items of known weights wi and values vi, i=1,2,….n
- Knapsack capacity W =10
- Item Weight Value Value/Weight

Item	Weight	Value	Value/Weight
1	4	$40	10
2	7	$42	6
3	5	$25	5
4	3	$12	4

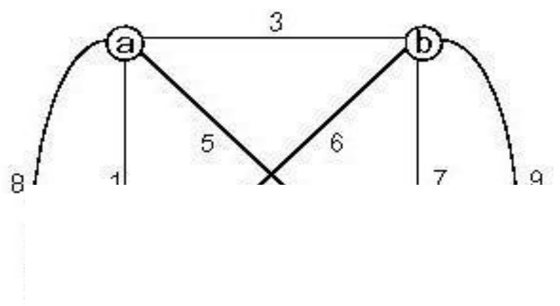
State space tree of knapsack problem

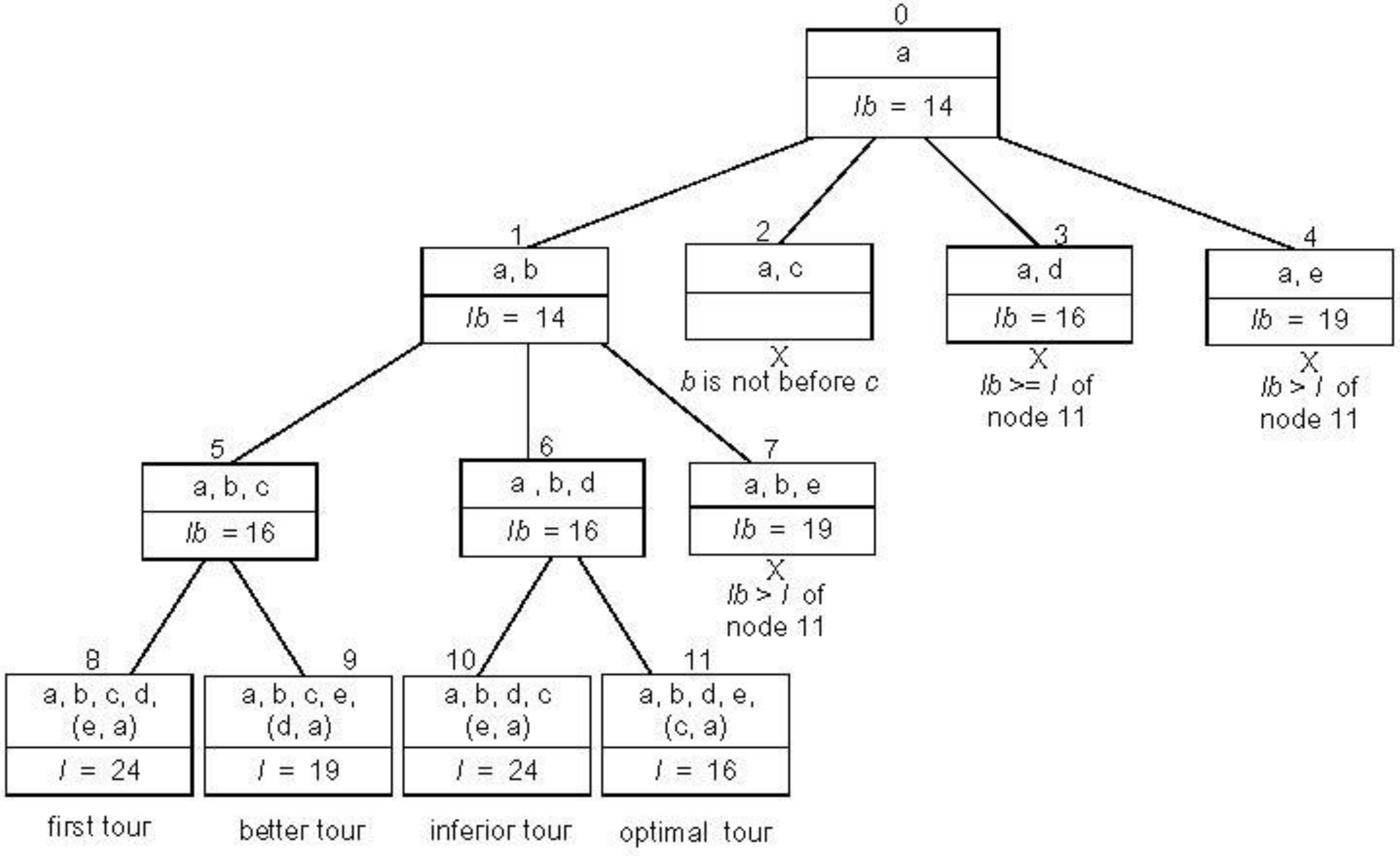


Traveling Salesman Problem

- For each city i, 1 <= i <=n, find the sum of the distances from city i to the two nearest cities.
- Compute the sum s of these n numbers
- Divide the result by 2
- If all the distances are integers, round up the result to the nearest integer
- lb = [s/2]

Example: Traveling Salesman Problem





UNIT - V

NP PROBLEMS & APPROXIMATION ALGORITHMS

P, NP and NP-Complete Problems

Problem Types: Optimization and Decision

- *Optimization problem*: find a solution that maximizes or minimizes some objective function

- Decision problem*:* answer yes/no to a question

Many problems have decision and optimization versions.

E.g.: traveling salesman problem
- *optimization*: find Hamiltonian cycle of minimum length
- *decision*: find Hamiltonian cycle of length $\leq m$

Decision problems are more convenient for formal investigation of their complexity.

Class *P*

P: the class of decision problems that are solvable in O(*p*(*n*)) time, where *p*(*n*) is a polynomial of problem's input size *n*

Examples:

- ☐searching
- ☐element uniqueness
- ☐graph connectivity
- ☐graph acyclicity
- ☐primality testing

Class *NP*

NP (*nondeterministic polynomial*): class of decision problems whose proposed solutions can be verified in polynomial time = solvable by a *nondeterministic polynomial algorithm*

A *nondeterministic polynomial algorithm* is an abstract two-stage procedure that:

- generates a random string purported to solve the problem
- checks whether this solution is correct in polynomial time

By definition, it solves the problem if it's capable of generating and verifying a solution on one of its tries

Example: CNF satisfiability

Problem: Is a boolean expression in its conjunctive normal form (CNF) satisfiable, i.e., are there values of its variables that makes it true?

This problem is in *NP*. Nondeterministic algorithm:

- ☐ Guess truth assignment
- ☐ Substitute the values into the CNF formula to see if it evaluates to true

Example: (A | ¬B | ¬C) & (A | B) & (¬B | ¬D | E) & (¬D | ¬E)

Truth assignments:

A	B	C	D	E
0	0	0	0	0
.	.	.		
1	1	1	1	1

Checking phase: O(*n*)

What problems are in *NP*?

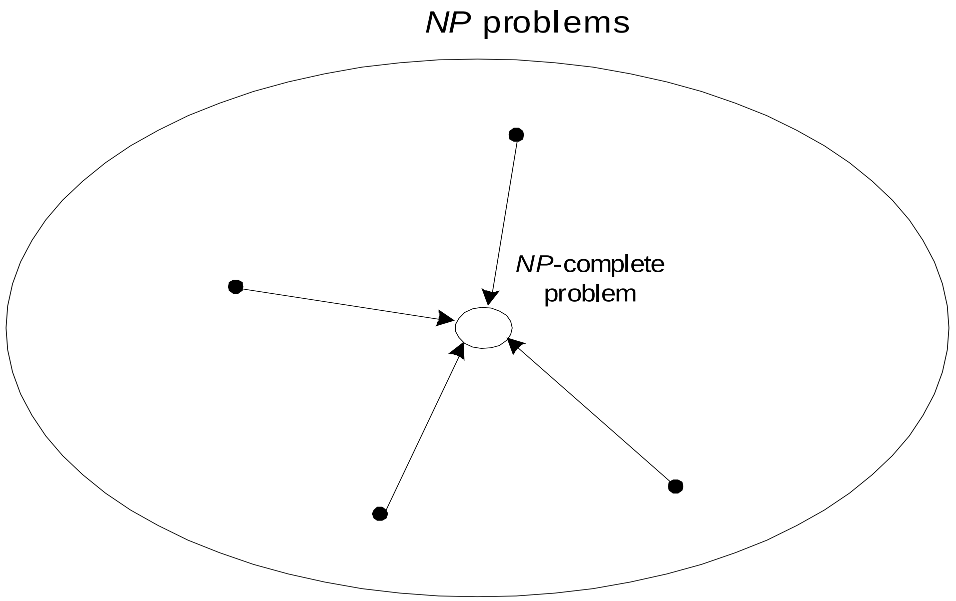
- Hamiltonian circuit existence
- Partition problem: Is it possible to partition a set of *n* integers into two disjoint subsets with the same sum?
- Decision versions of TSP, knapsack problem, graph coloring, and many other combinatorial optimization problems. (Few exceptions include: MST, shortest paths)

- All the problems in *P* can also be solved in this manner (but no guessing is necessary), so we

NP-Complete Problems

A decision problem *D* is <u>*NP*-complete</u> if it's as hard as any problem in *NP*, i.e.,

- *D* is in *NP*
- every problem in *NP* is polynomial-time reducible to *D*

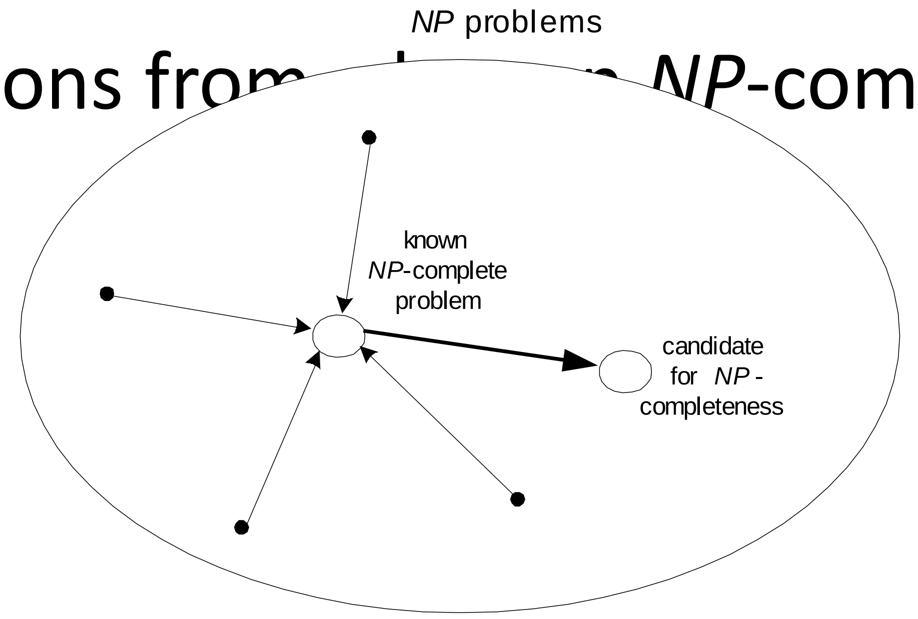


Cook's theorem (1971): CNF-sat is *NP*-complete

NP-Complete Problems (cont.)

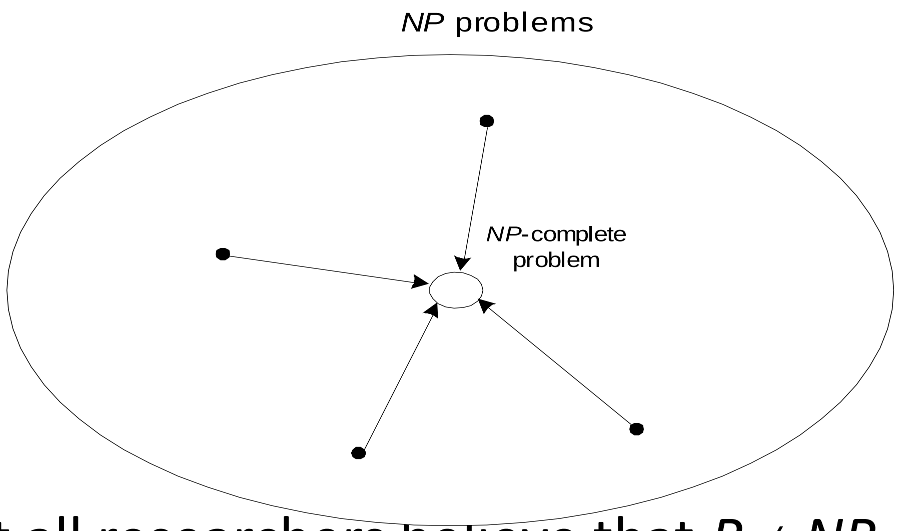
Other *NP*-complete problems obtained through polynomial-

time reductions from [illegible] *NP*-complete problem



P = *NP* ? Dilemma Revisited

- *P* = *NP* would imply that every problem in *NP,* including all *NP*-complete problems, could be solved in polynomial time
- If a polynomial-time algorithm for just one *NP*-complete problem is discovered, then every problem in *NP* can be solved in polynomial time, i.e., *P* = *NP*



- Most but not all researchers believe that $P \neq NP$, i.e. *P* is a proper subset of *NP*