

Design and Analysis of Algorithm

PPTS prepared using Text Book

Ellis Horowitz, SatrajSahni and S Rajasekharam,
Fundamentals of Computer Algorithms

Dr. Y. Sri Lalitha

Motivation

- Design and Analysis of Algorithms is useful in the study of
 - Computer Science
 - Information Technology
 - Natural Sciences
 - Industry etc
- It is a study to understand different algorithm **paradigms**, **analyzing** algorithms performance in worst, average and best cases and **its application** while designing new algorithms.
- **The Course Emphasis on**
 - How to compute Best, Average and Worst performance algorithms
 - How to find Optimal Solutions to a given problem.
 - How to find Optimal path in a Tour/Network Traffic and so on
 - How to find best ways to solve a given problem.
- Outcome of the course is, the learner will be able to decide which method is best to solve a given problem.

Syllabus Course code : GR22A2077

- **UNIT I Introduction:** Definition of an algorithm, properties of an Algorithm, performance analysis-space complexity & time complexity, **Asymptotic notations:** big oh notation, omega notation, theta notation, little oh notation & little omega notation. Amortized Analysis
- **UNIT II Disjoint sets:** Disjoint set Representation, Operations, union and find algorithms. **Divide and conquer:** General method, applications, binary search, merge sort, strassen's matrix multiplication.
- **UNIT III Dynamic programming :** General method, applications, matrix chain multiplication, optimal binary search trees, 0/1 knapsack problem All pairs shortest path problem, travelling salesperson problem, reliability design, optimal rod-cutting-Top down approach and bottom up approach.

Syllabus

- **UNIT IV Greedy method:** General method, applications-- job sequencing with deadlines, 0/1 knapsack problem, minimum cost spanning trees, single source shortest path problem, activity selection problem. **Backtracking:** General method, applications, n-queen problem, sum of subsets problem, graph coloring, Hamiltonian cycles.
- **UNIT V Branch and Bound:** General method, applications, travelling salesperson problem, 0/1 knapsack problem: LC branch and bound solution, FIFO branch and bound solution. Complexity Classes: non-deterministic algorithms, deterministic algorithms, relationship between P, NP-hard and NP-complete problems, circuit satisfiability problem, 3-CNF satisfiability.

- **TEXT BOOKS**

1. Ellis Horowitz, SatrajSahni and S Rajasekharam, Fundamentals of Computer Algorithms, Galgotia publishers
2. T H Cormen, C E Leiserson, and R L Rivest, Introduction to Algorithms, 3rdEdn, Pearson Education
3. Goodrich, Michael T. & Roberto Tamassia, Algorithm Design, Wiley Singapore Edition, 2002.

Fundamentals of Algorithm Analysis and Design

- Algorithmic is a branch of computer science that consists of designing and analyzing computer instructions.
- The “design” pertain to
 - The description of algorithm at an abstract level by means of a pseudo language, and Proof of correctness that is, the algorithm solves the given problem in all cases.
- The “analysis” deals with performance evaluation (complexity analysis).
- This course is all about the mathematical theory behind the **design of good programs**

Algorithm - Definition

Algorithm can be defined in many ways.

- An algorithm is a set of rules for carrying out calculation either by hand or on a machine.
- An algorithm is a finite step-by-step procedure to achieve a required result.
- An algorithm is a sequence of computational steps that transform the input into the output.
- An algorithm is an abstraction of a program to be executed on a physical machine (model of Computation).
- Are mathematical entities, which can be thought of as running on some sort of *idealized computer with an infinite random access memory*

Algorithm: - properties

- Algorithm: Finite set of instructions that, if followed, accomplishes a particular task.
- Described in natural language / pseudo-code / diagrams / etc.
- Algorithms must satisfy the following Criteria :
 - Input: Zero or more quantities (externally produced)
 - Output: One or more quantities
 - Definiteness: Clarity, precision of each instruction
 - Finiteness: The algorithm has to stop after a finite (may be very large) number of steps
 - Effectiveness: Each instruction has to be basic enough and feasible

Algorithm: - properties

- Definiteness

- 5/0 or add 6 or 7 to variable x are not permissible, the instructions should be crisp and clear.

- Finiteness

- We assume that an algorithm terminates after a finite set of steps.
- Chess game : An algorithm can be devised to check whether any given position in the game is winning position.

- Effectiveness

- Each step should be effective, such that taking a paper and pen should be able to solve in a finite time. Integer arithmetic is effective whereas arithmetic on real values is not effective.
- Algorithms that are definite and effective are called Computational Procedures(CP).
 - Operating System is an example of Computational Procedures that do not terminate, but continues to wait for a new job.
 - This course restricts to CPs that always terminates.

Why should we Study Algorithms

- There are four Aspects of Algorithm Study
 - How to device algorithms ?
 - Mastering different Design Strategies – helps in designing new and useful algorithms.
 - How to validate algorithms ?
 - Algorithm Validation
 - Assures that algorithm works correctly independent of Programming Language.
 - Program Proving or Program Verification
 - Proof of program correctness implies each statement of programming language be precisely defined and all basic operations be proved correct.
 - How to analyze algorithms ?
 - Determines Computing Time and Storage
 - Allows to judge the efficiency of one algorithm over another.
 - How well the algorithm perform in best, average and worst cases.
 - Allows to predict the efficiency constraints of the algorithm.

Why should we Study Algorithms

- How to Test a program

- Debugging and Profiling (Performance Measurement)
- Debugging is the process of executing Programs on Sample datasets to determine whether faulty results occur, if so Correct them.
 - *Debugging can only point to the presence of error but not to their absence.*
 - Testing is happening on Sample data and not on whole data.
- Profiling is the process of executing correct program on datasets and measuring the time and space it takes to compute the results.
 - *Profiling points out logical places to perform useful optimizations*

Algorithm Specification

- *How to describe algorithms independent of a programming language?*
- An algorithm can be written in many ways
 - In English like Language
 - Graphical Representation (Flow chart).
 - Pseudocode

The course uses a pseudocode representation much like a c-language.

- Pseudo-Code = a description of an algorithm that is
 - more structured than usual prose but
 - less formal than a programming language
 - (Or diagrams)

Algorithm Specifications

- Expressions: use standard mathematical symbols
 - use `:=` for assignment (`=` in C/C++)
 - use `=` for the equality relationship (`==` in C/C++)
 - array indexing: **A[i]**
 - Elements of multidimensional array are denoted as **A[i,j]**
- Method Declarations: -**Algorithm** name(*param1, param2*)
- Methods
 - calls: `method(args)`
 - returns: **return** value
- Use **comments** indicated by `//` like in c-language
- Instructions have to be basic enough and feasible!

Looping and Decision statement

- Programming Constructs:

- while-loops

```
        while ... do  
            • While(condition ) do  
            {  
                stmt 1  
                ..  
                Stmt n  
            }
```

while ... do

{

stmt 1

....

stmt n

}until ((condition))

- for-loop:

```
for i:=1 to n do  
{  
    print i  
    sum = Sum + I  
}
```

for ... do

decision structures: if .. then .. [else ..]

if(condition) then
stmt1

if(condition) then
stmt 1

else

stmt2

repeat-loops:

Repeat

repeat ... until ...

Example : Find the maximum element of an array

Algorithm arrayMax(A, n){

// *Input*: An array A storing n integers.

// *Output*: The maximum element in A.

currentMax := A[0]

for *I* := 1 to *n* -1 **do**

if *currentMax* < A[i] **then** *currentMax* := A[i]

return *currentMax*

}

Exercise : Write a Sorting algorithm using the specification mentioned in todays class.

Selection Sort

- Algorithm SelectionSort(a,n)

```
// sort elements of array[1..n] into increasing order
```

```
{
```

```
    for i:=1 to n do
```

```
{
```

```
        minInd := i
```

```
        for k := i+1 to n do
```

```
            if( a[k] < a[minInd]) then minInd:= k;
```

```
            t:=a[i];
```

```
            a[i]:= a[minInd]; a[minInd] := t;
```

```
}
```

```
}
```

3. Algorithm Analysis

- Performance evaluation of a task is loosely divided into two major phases:

1. *Priori* estimates also called as Performance Analysis :

Is to obtain estimates of time and space requirement of a task (**algorithm**) **independent of machine**.

2. *Posteriori Testing* also called as Performance Measurement:

Is to obtain the **actual** space and time requirements of a task (**program**) **Machine dependent**.

- **Space complexity** : The amount of memory required by an algorithm to run to completion.
 - How much space is required
- **Time complexity** : The amount of time required by an algorithm to run to completion.
 - How much time is required

Performance Analysis – Space (1/5)

- The space needed by a program is seen to be the sum of the following components:
 - A **fixed** space requirement:
 - The required memory that is independent of the instance characteristics of the program.
 - e.g., instruction (code) space, simple variables, fixed-size structured variables constants return address and so on.
- A **variable** space requirement:
 - Memory that is dependent upon the instance characteristics of the program.
 - e.g., the recursion stack space.

Instance characteristics of a program:

number and size of the program's inputs and outputs,
a specific problem. e.g., factorial(3)

Performance Analysis – Space (2/5)

- The space requirement $S(P)$ of a program P can be written as:

- $$S(P) = c + S_p \text{ (instance characteristics).}$$

constant (or fixed memory requirement)

variable memory requirement,
depends on particular instance characteristics.

- We usually concentrate on estimating S_p (instance characteristics).

- For a given problem, we shall need to first determine which instance characteristics to use to measure the space requirements.

Performance Analysis – Space (3/5)

```
Algorithm Abc( a, b, c)
{
    return a + b + b * c;
}
```

- The instance is characterized by specific values of a,b and c
 - $S_{Abc} = 0$, in terms of instance characteristics
 - $S_{Abc} \geq 4$ units, in terms of units of memory but we indicate it as Constant O(1) independent of instance characteristics

Example: Algorithm *sum* (*a, n*)

```
{   s := 0;
    for i = 0 to n do
        s+ = a[i];
    return s;
}
```

Space? n words for array $a[]$, if array is passed, one word for each variables $n, i, s, S_{sum} \geq (n+4)$

Performance Analysis Space Complexity (4/5)

```
Algorithm: factIter( n )
{ fact := 1
  for i:=1 to n do
    fact := fact * n
  return fact
}
```

```
Algorithm factRecur(n) then
{
  if (n == 0)
    return 1;
  else
    return n * factRecur(n-1);
}
```

- The instances are characterized by n .
 - Suppose each recursion stack space requires 2 bytes, s_p would be $2(n+1)$ bytes.

Performance Analysis Space Complexity (4/5)

- Algorithm Rsum(a,n){
 if (n <= 0) then
 return 0;
 return Rsum(a, n-1) + a[n];
}

The instances are characterized by n.

The **recursion stack space** includes space for the formal parameters, the local variables, and the return address.

It requires 3 words, (for n, pointer to a and return address)
Depth of recursion stack is $n+1$. $S_{Rsum} \geq 3(n+1)$

Algorithm Transpose

```
1  Algorithm Transpose( $a, n$ )
2  {
3      for  $i := 1$  to  $n - 1$  do
4          for  $j := i + 1$  to  $n$  do
5              {
6                   $t := a[i, j]; a[i, j] := a[j, i]; a[j, i] := t;$ 
7              }
8  }
```

Performance Analysis — Space Complexity (5/5)

- More complex than time complexity.
- It strongly depends on data structures design.
 - It affects the computing time complexity.
- Significant
- In general-purpose computing platform:
 - Memory, cache, secondary storage.
- In embedded system:
 - Limited memory (SDRAM, FLASH (ROM)).
 - No secondary storage.

Performance Analysis — Time Complexity (1/4)

- The time, $T(P)$ taken by a program P is the sum of the compile time and the execution time.
 - We assume that a compiled program will be run several times without recompilation.
 - Consequently, we shall concern ourselves with just the execution time of a program.
 - Denoted by t_p (*instance characteristics*).
 - we shall need to first determine which instance characteristics to use to measure the time complexity.
 - If we knew the characteristics of the compiler to be used, we could obtain an expression for $t_p(n)$ of the form:

$$t_p(n) = c_a ADD(n) + c_s SUB(n) + c_m MUL(n) + c_d DIV(n) + \dots$$



Time need for an addition Number of additions

Performance Analysis —Time Complexity (2/4)

- Obtaining such a formula is an **impossible** task!!
 - The time needed for an addition often depends on the actual numbers being added.
 - In a multiuser system, the execution time will depend on many other factors.
 - E.g., the number of other running programs.
- We can go one step further and count only the **number of program steps**.
- A **program step**:
 - Is defined as a syntactically or semantically meaningful segment of a program. Its execution time is independent of the instance characteristics.
 - Example: $a = b + c + b * c + (a + b)/c + 3.14;$
 $a = b + c$

Step counts in Algorithm

- Comments count as zero steps
- Assignment statement which does not involve any call to other algorithms count one.
- Control parts of for, while and repeat are counted.
 - Ex : for i:= <expr> to <expr1> do
 - While (i <= expr1) do

The step count is equal to the no of values assigned to looping variable.

Let N be the no. of values assigned to i
then the control statement step count is N

Remaining steps in the control statements (for, while)
will have step count = N -1 for each statement in the control part of
(for, while)

- Return statement counts one step.

Two ways to compute the time complexity using Program steps of an algorithm

1. Introduce a global count variable in the algorithm.
Each time a statement in the original program is executed this variable is incremented by its step count.
2. Build a table in which we list the total no. of steps contributed by each statement.

Often arrived by no. of steps per execution denoted by (s/e) and the total no. of times (frequency) each statement is executed.

Sum of n numbers using step count

```
1  Algorithm Sum( $a, n$ )
2  {
3       $s := 0.0$ ;
4       $count := count + 1$ ; // count is global; it is initially zero.
5      for  $i := 1$  to  $n$  do
6      {
7           $count := count + 1$ ; // For for
8           $s := s + a[i]$ ;  $count := count + 1$ ; // For assignment
9      }
10      $count := count + 1$ ; // For last time of for
11      $count := count + 1$ ; // For the return
12     return  $s$ ;
13 }
```

Algorithm 1.8 Algorithm 1.6 with count statements added

Step count approach for Matrix Sum

```
1  Algorithm Add( $a, b, c, m, n$ )
2  {
3      for  $i := 1$  to  $m$  do
4          for  $j := 1$  to  $n$  do
5               $c[i, j] := a[i, j] + b[i, j];$ 
6  }
```

Algorithm 1.11 Matrix addition

Matrix sum counting statements

Approach

```
1  Algorithm Add( $a, b, c, m, n$ )
2  {
3      for  $i := 1$  to  $m$  do
4          {
5              count := count + 1; // For 'for i'
6              for  $j := 1$  to  $n$  do
7                  {
8                      count := count + 1; // For 'for j'
9                       $c[i, j] := a[i, j] + b[i, j];$ 
10                     count := count + 1; // For the assignment
11                 }
12                 count := count + 1; // For loop initialization and
13                             // last time of 'for j'
14             }
15             count := count + 1; // For loop initialization and
16                         // last time of 'for i'
17 }
```

Recursive Sum of n nos.

```
1  Algorithm RSum(a,n)
2  {
3      count := count + 1; // For the if conditional
4      if (n ≤ 0) then
5      {
6          count := count + 1; // For the return
7          return 0.0;
8      }
9      else
10     {
11         count := count + 1; // For the addition, function
12                         // invocation and return
13         return RSum(a,n - 1) + a[n];
14     }
15 }
```

Rsum Step count

Analyzing a Recursive algorithm for its step count, we obtain a recursive formula , Known as recurrence relation.

$$t_{\text{RSum}}(n) = \begin{cases} 2 & \text{if } n = 0 \\ 2 + t_{\text{RSum}}(n - 1) & \text{if } n > 0 \end{cases}$$

$$\begin{aligned} t_{\text{RSum}}(n) &= 2 + t_{\text{RSum}}(n - 1) \\ &= 2 + 2 + t_{\text{RSum}}(n - 2) \\ &= 2(2) + t_{\text{RSum}}(n - 2) \\ &\vdots \\ &= n(2) + t_{\text{RSum}}(0) \\ &= 2n + 2, \quad n \geq 0 \end{aligned}$$

step count for **RSum** (Algorithm 1.7) is $2n + 2$.

Steps/Execution approach

Statement	s/e	frequency	total steps
1 Algorithm Sum(a, n)	0	–	0
2 {	0	–	0
3 $s := 0.0;$	1	1	1
4 for $i := 1$ to n do	1	$n + 1$	$n + 1$
5 $s := s + a[i];$	1	n	n
6 return $s;$	1	1	1
7 }	0	–	0
Total			$2n + 3$

s/e approach for Recursive sum of n nos.

Statement	s/e	frequency		total steps	
		$n = 0$	$n > 0$	$n = 0$	$n > 0$
1 Algorithm RSum(a, n)	0	—	—	0	0
2 {					
3 if ($n \leq 0$) then	1	1	1	1	1
4 return 0.0;	1	1	0	1	0
5 else return					
6 RSum($a, n - 1$) + $a[n]$;	$1 + x$	0	1	0	$1 + x$
7 }	0	—	—	0	0
Total				2	$2 + x$

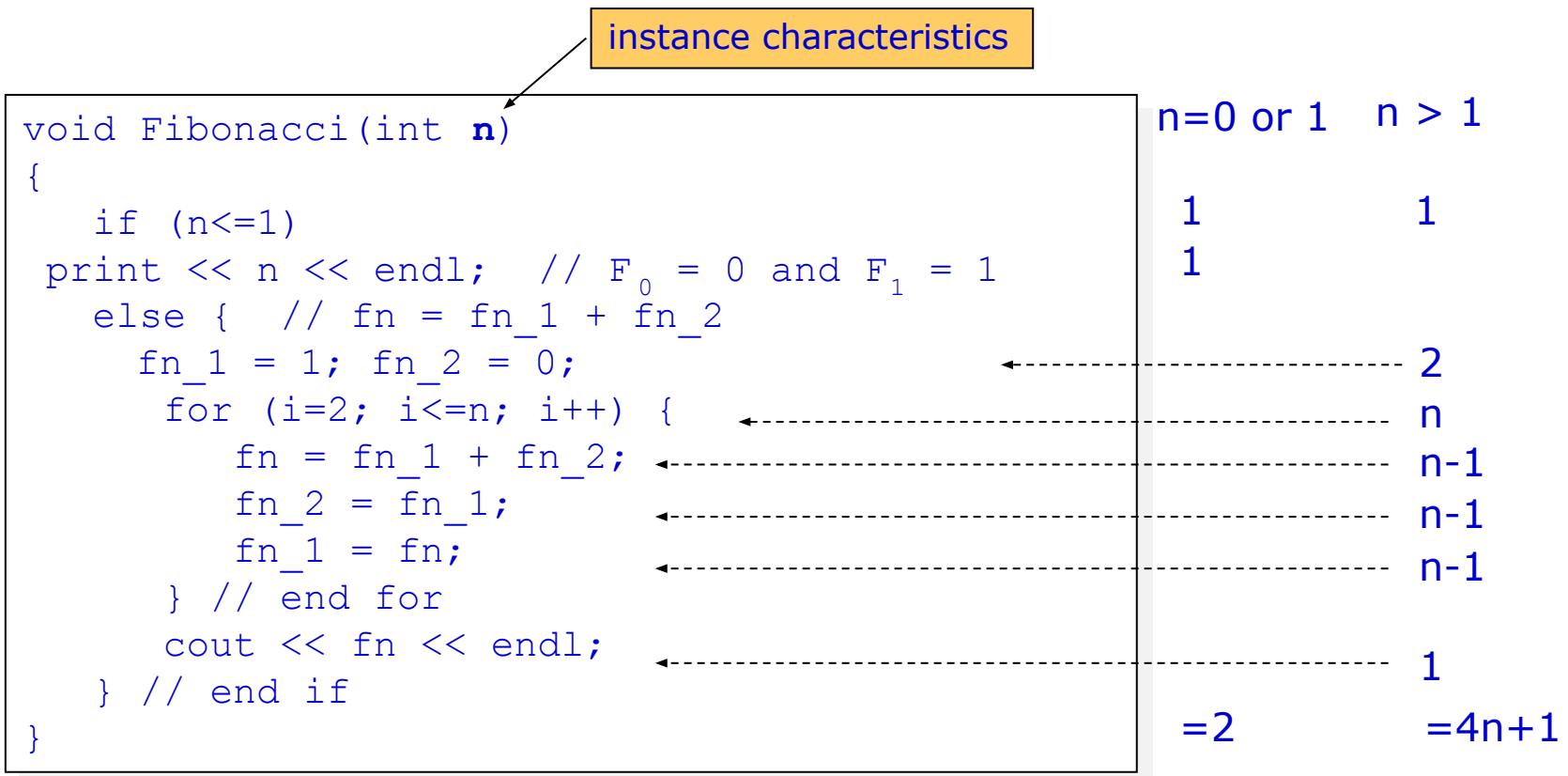
$$x = t_{\text{RSum}}(n - 1)$$

s/e for Matrix addition.

Statement	s/e	frequency	total steps
1 Algorithm Add(a, b, c, m, n)	0	–	0
2 {	0	–	0
3 for $i := 1$ to m do	1	$m + 1$	$m + 1$
4 for $j := 1$ to n do	1	$m(n + 1)$	$mn + m$
5 $c[i, j] := a[i, j] + b[i, j];$	1	mn	mn
6 }	0	–	0
Total			$2mn + 2m + 1$

Performance Analysis — Time Complexity (4/4)

- Example 2:



Performance Analysis

- Two important reasons to determine frequency and step counts of an algorithm
 1. To compare the time complexities of two programs that compute the same function
 2. To predict the growth in run time as the instance characteristic changes
- Determining the exact step count of a program is a very difficult task.
 - The notion of a step is itself **inexact**.
 - E.g., $x = y$ and $x = y + (x/y) + (x*y*z)$ all count as one step!!
 - $t_p(n) = 100n + 10 \Leftrightarrow 75n \leq t_p(n) \leq 120n.$
- We are interested in precise notation for characterizing running-time differences that are likely to be significant across different platforms and different implementations of the algorithms

Asymptotic Time Complexity (1)

- Two programs have complexities $c_1 n^2 + c_2 n$ and $c_3 n$, respectively
- The program with complexity $c_3 n$ will be faster than the one with complexity $c_1 n^2 + c_2 n$ for sufficiently large values of n
- For small values of n , either program could be faster \square depends on the values of c_1 , c_2 and c_3
- If $c_1 = 1$, $c_2 = 2$, $c_3 = 100$, then $c_1 n^2 + c_2 n \leq c_3 n$ for $n \leq 98$ and $c_1 n^2 + c_2 n > c_3 n$ for $n > 98$
- What if $c_1 = 1$, $c_2 = 2$, and $c_3 = 3$?

Asymptotic Analysis

- In Asymptotic Analysis, the performance of an algorithm is evaluated in terms of **input size** (we don't measure the actual running time).
- We Calculate how does the **time/space varies with varying input size**
- In mathematical analysis, **asymptotic analysis**, also known as **asymptotics**, is a method of describing limiting behavior.
 - For Ex: suppose that we are interested in the properties of a function $f(n)$ as n becomes very large.
 - If $f(n) = n^2 + 3n$, then as n becomes very large, the term $3n$ becomes insignificant compared to n^2 .
 - The function $f(n)$ is said to be "*asymptotically equivalent to n^2 , as $n \rightarrow \infty$* ". This is often written symbolically as $f(n) \sim n^2$, which is read as " $f(n)$ is asymptotic to n^2 ".

Asymptotic Notations

- **Asymptotic Notations** are the expressions that are used to represent the time/space limiting behaviors (Complexities) of an algorithm.
 - It is concerned with how the running time of an algorithm **behaves** as the input size varies.
 - It allows us to ignore small input sizes, constant factors, lower-order terms in polynomials, and so forth.
 - If $f(n) = n^2 + 3n$, then as n becomes very large, the term $3n$ becomes insignificant compared to n^2 .

Asymptotic Notations

- Methods to Measure Efficiency
- Let $f(n)$ and $g(n)$ be functions that map positive integers to positive real numbers.
- **Big Oh (O)** notation provides a **Least upper bound/tight upper bound** for the function f
 - $O(g(n))$: class of functions $f(n)$ that grow no faster than $g(n)$
- **Little oh (o)** defines a **upper bound / loose upper bound** for the function f .
- **Theta (Θ)** notation is used when an algorithm can be **bounded both from above and below** by the same function
 - $\Theta(g(n))$: class of functions $f(n)$ that grow at same rate as $g(n)$
- **Omega (Ω)** notation provides a **Greatest lower-bound/tight lower-bound** for the function f
 - $\Omega(g(n))$: class of functions $f(n)$ that grow at least as fast as $g(n)$
- **Little omega (ω)** defines a **lower bound / loose lower bound** for the function f

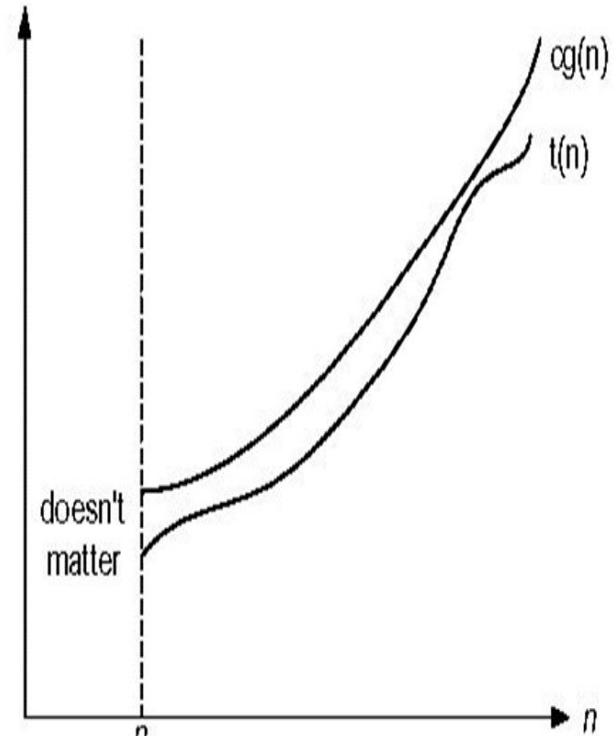
Big Oh Notation

- Definition : $f(n) = O(g(n))$ (read as “ $f(n)$ is Big Oh of $g(n)$ ”)

iff there exists two positive constants c and n_0 such that $f(n) \leq c * g(n)$ for all $n, n \geq n_0$.

- Show that $2n+3 = O(n)$

- If we show that $f(n) \leq c * g(n)$ for all values of n where $n \geq n_0$ then $f(n) = O(g(n))$.
- The statement $f(n) = O(g(n))$ states that $g(n)$ is an **upper bound** on the value of $f(n)$ for all $n, n \geq n_0$.



e.g., $f(n)$ of bubble sort lies in the complexity class $O(n^2)$.

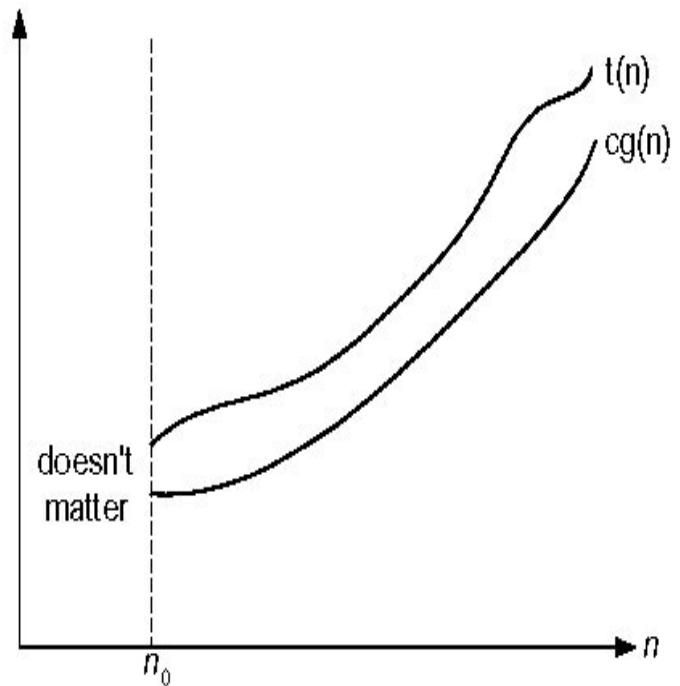
Omega (Ω) Notation

- Definition : $f(n) = \Omega(g(n))$ (read as “ $f(n)$ is omega of $g(n)$ ”) iff there exists positive constants c and n_0 such that $f(n) \geq cg(n)$ for all $n, n \geq n_0$.

Ex : bubble sort has to look at all elements which takes $\Omega(n)$

Let $f(n) = 2n^2 + 7n - 10$ and $g(n) = n^2$. we have $2n^2 + 7n - 10 \geq c \cdot n^2$, with $c = 1$ and $n_0 \geq 2$ thus $f(n) = \Omega(g(n))$.

$f(n)$ lies in $\Omega(n^2)$, i.e., it grows at least quadratically.



Theta (Θ) Notation (Average Case)

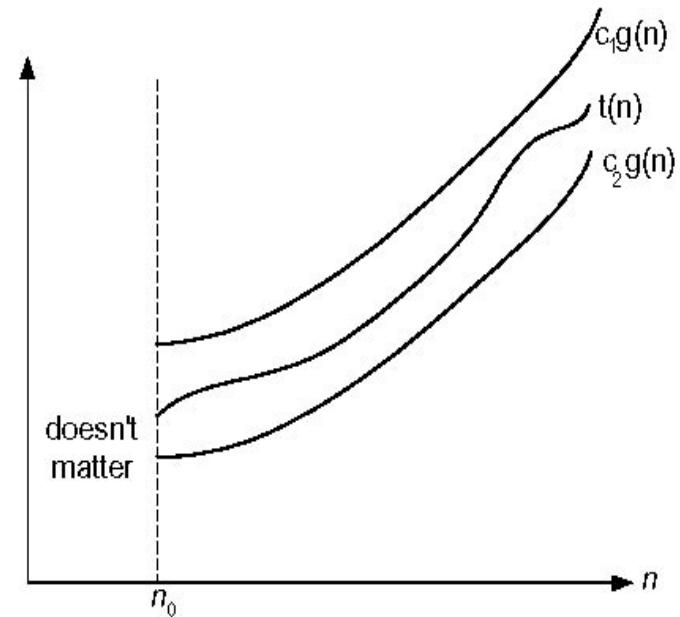
- Used when the function f can be **bounded both from above and below** by the same function g .
- Definition:** $f(n) = \Theta(g(n))$ (read as “ $f(n)$ is theta of $g(n)$ ”) iff there exists positive constants c_1, c_2 and n_0 such that $c_1 g(n) \leq f(n) \leq c_2 g(n)$ for all $n, n \geq n_0$.

That is, f lies between c_1 times the function g and c_2 times the function g , except possibly when n is smaller than n_0 .
 $3n+2 = \Theta(n)$ as $3n+2 \geq 3n$ for all $n \geq 2$ and $3n+2 \leq 4n$ for all $n \geq 2$.

$$3n \leq 3n+2 \leq 4n$$

Therefore $3n + 2 = \Theta(n)$

The theta notation is more precise than both the “big oh” and omega notations. Provides both an upper and lower bound on $f(n)$.



Theta notation

- $2n \leq 2n+3 \leq 3n$

LHS : $2n \leq 2n+3$

$$2 \leq 2 + \underline{3}$$

RHS : $2n+3 \leq 3n$

$$2 + \underline{3} \leq 3$$

	n
n=1	5
2	3.5
3	2

...	...
∞	2

$c_1 = 2$ and $n_0 \geq 1$

	n		n
n=1	5	n=1	5
2	3.5	2	3.5
3	2	3	2
...
∞	2	∞	2

$c_2 = 3$ and $n_0 \geq 3$

Therefore for $c_1 = 2, c_2 = 3, n_0 = 3$ $2n+3 = \Theta(n)$.

Little oh and little omega

- Definition [**Little oh**]: Definition: $f(n) = o(g(n))$ (read as “ $f(n)$ is little oh of $g(n)$ ”) iff there exists positive constants c and n_0 such that $f(n) < c \cdot g(n)$ for all n , $n \geq n_0$.

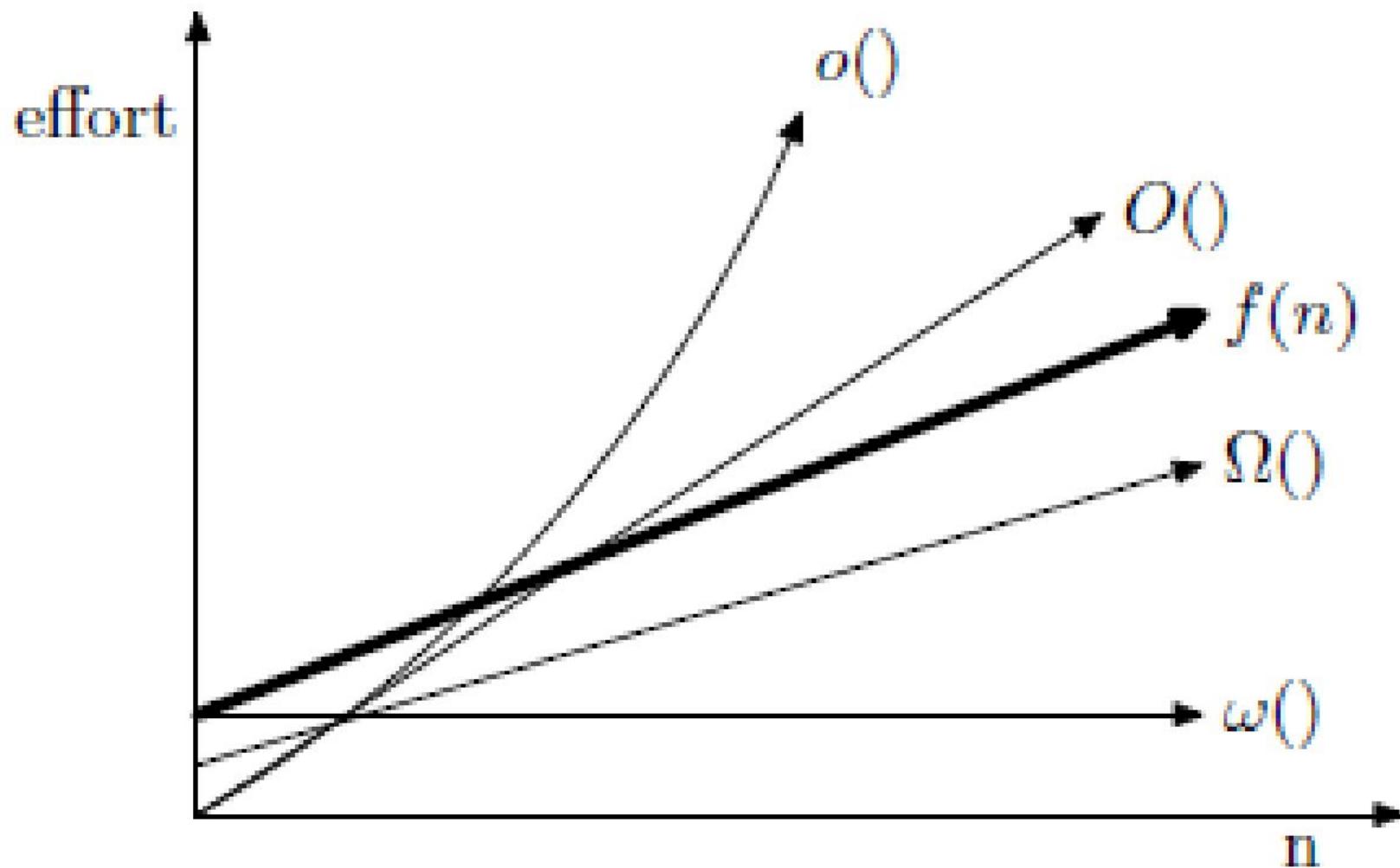
$$\lim_{n \rightarrow \infty} \frac{f(n)}{g(n)} = 0$$

- $3n+2 = o(n^2)$ since $\lim_{n \rightarrow \infty} (3n + 2)/n^2 = 0$
- $3n+2 = o(n \log n)$

- Definition [**Little omega**]: $f(n) = \omega(g(n))$ (read as “ $f(n)$ is little omega of $g(n)$ ”) iff there exists positive constants c and n_0 such that $f(n) > c \cdot g(n)$ for all n , $n \geq n_0$.

$$\lim_{n \rightarrow \infty} \frac{g(n)}{f(n)} = 0$$

Relation between different notations



Examples

Statement	s/e	frequency	total steps
1 Algorithm Sum(a, n)	0	–	$\Theta(0)$
2 {	0	–	$\Theta(0)$
3 $s := 0.0;$	1	1	$\Theta(1)$
4 for $i := 1$ to n do	1	$n + 1$	$\Theta(n)$
5 $s := s + a[i];$	1	n	$\Theta(n)$
6 return $s;$	1	1	$\Theta(1)$
7 }	0	–	$\Theta(0)$
Total			$\Theta(n)$

Statement	s/e	frequency		total steps	
		$n = 0$	$n > 0$	$n = 0$	$n > 0$
1 Algorithm RSum(a, n)	0	—	—	0	$\Theta(0)$
2 {	0	—	—	0	$\Theta(0)$
3 if ($n \leq 0$) then	1	1	1	1	$\Theta(1)$
4 return 0.0;	1	1	0	1	$\Theta(0)$
5 else return					
6 RSum($a, n - 1$) + $a[n]$);	$1 + x$	0	1	0	$\Theta(1 + x)$
7 }	0	—	—	0	$\Theta(0)$
Total				2	$\Theta(1 + x)$

$$x = t_{\text{RSum}}(n - 1)$$

Performance Analysis —Asymptotic Notation

- Example 3: Matrix addition.
 - a , b , and c are matrix of $m \times n$.
 - $c = a + b$, that is, $c_{ij} = a_{ij} + b_{ij}$.

instance characteristics

```
void add(int **a, int **b, int **c, int m, int n)
{
    for (int i=0; i<m; i++)
        for( int j=0; j<n; j++)
            c[i][j] = a[i][j] + b[i][j];
}
```

$\Theta(m)$

$\Theta(mn)$

$\Theta(mn)$

$\Theta(mn)$

Common Growth Rate Functions

- 1 (**constant**):
 - growth is independent of the problem size n.
- $\log_2 N$ (**logarithmic**):
 - growth increases slowly compared to the problem size (binary search)
- N (**linear**):
 - directly proportional to the size of the problem.
- $N * \log_2 N$ (**$n \log n$**):
 - typical of some divide and conquer approaches (merge sort)
- N^2 (**quadratic**):
 - typical in nested loops
- N^3 (**cubic**):
 - more nested loops
- 2^N (**exponential**):
 - growth is extremely rapid and possibly impractical.
- Hierarchy of growth rate functions:
 $1 < \log n < n < n \log n < n^2 < n^3 < 2^n < 3^n < n! < n^n$

Practical Complexities

log n	n	n log n	n^2	n^3	2^n
0	1	0	1	1	2
1	2	2	4	8	4
2	4	8	16	64	16
3	8	24	64	512	256
4	16	64	256	4096	65536
5	32	160	1024	32768	4294967296

Overly complex programs may not be practical given the computing power of the system.

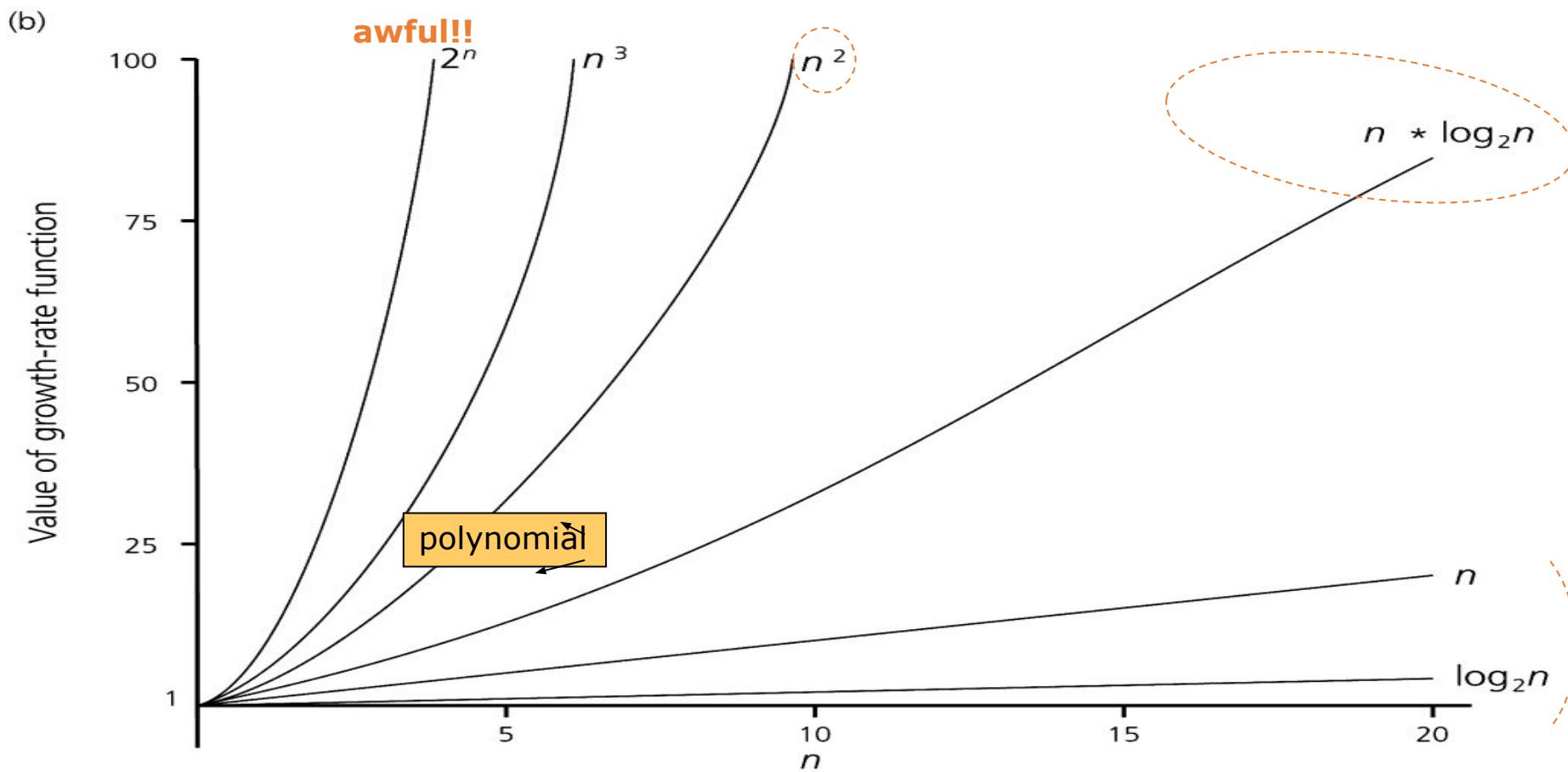
Performance Analysis — Practical Complexities

- A computer performing 1 billion steps per second.

n	$f(n)$					2^n
	n	$n \log n$	n^2	n^3		
10	.01μs	.03μs	.1μs	1μs		1μs
20	.02μs	.09μs	.4μs	8μs		1ms
30	.03μs	.15μs	.9μs	27μs		1s
40	.04μs	.21μs	1.6μs	64μs		18m
50	.05μs	.28μs	2.5μs	125μs		13d
100	.10μs	.66μs	10μs	1ms		$4*10^{13}y$

Performance Analysis — Practical Complexities

- The time complexity of a program can be used to determine how the time requirements vary as the instance characteristics change.



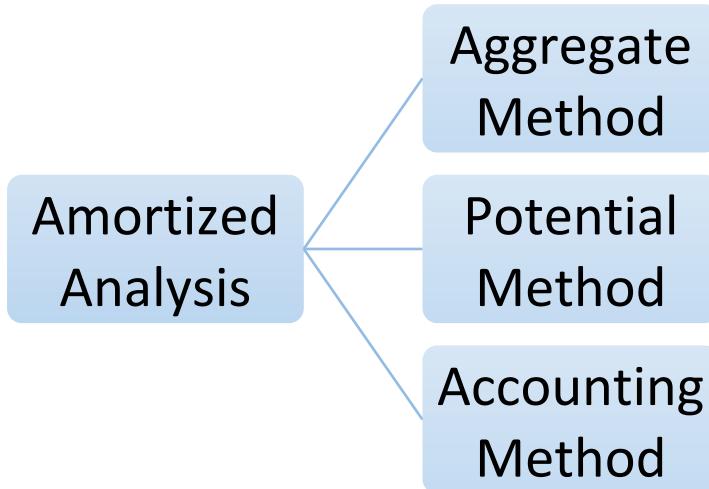
Introduction to Amortized Analysis

[Amortized Analysis](#) is used for algorithms where an occasional operation is very slow, but most of the other operations are faster. In Amortized Analysis, we analyze a sequence of operations and guarantee a worst-case average time that is lower than the worst-case time of a particularly expensive operation.

For Example, Amortized cost of a sequence of operations can be seen as expenses of a salaried person. The average monthly expense of the person is less than or equal to the salary, but the person can spend more money in a particular month by buying a car or something. In other months, he or she saves money for the expensive month.

The example data structures whose operations are analyzed using Amortized Analysis are Hash Tables, Disjoint Sets, and Splay Trees.

Amortized analysis is a technique used in computer science to analyze the average-case time complexity of algorithms that perform a sequence of operations, where some operations may be more expensive than others. The idea is to spread the cost of these expensive operations over multiple operations, so that the average cost of each operation is constant or less.



Consider the dynamic array data structure that can grow or shrink dynamically as elements are added or removed. The cost of growing the array is proportional to the size of the array, which can be expensive. However, if we amortize the cost of growing the array over several insertions, the average cost of each insertion becomes constant or less.

Amortized analysis provides a useful way to analyze algorithms that perform a sequence of operations where some operations are more expensive than others, as it provides a guaranteed upper bound on the average time complexity of each operation, rather than the worst-case time complexity.

Instead of analyzing the worst-case time complexity of an algorithm, which gives an upper bound on the running time of a single operation, amortized analysis provides an average-case analysis of the algorithm by considering the cost of several operations performed over time.

The key idea behind amortized analysis is to spread the cost of an expensive operation over several operations. For example, consider a dynamic array data structure that is resized when it runs out of space. The cost of resizing the array is expensive, but it can be amortized over several insertions into the array, so that the average time complexity of an insertion operation is constant.

The amortized analysis doesn't involve probability. There is also another different notion of average-case running time where algorithms use randomization to make them faster and the expected running time is faster than the worst-case running time. These algorithms are analyzed using Randomized Analysis. Examples of these algorithms are Randomized Quick Sort, Quick Select and Hashing. We will soon be covering Randomized analysis in a different post.

Aggregate Method

Amortized analysis is useful for designing efficient algorithms for data structures such as dynamic arrays, priority queues, and disjoint-set data structures. It provides a guarantee that the average-case time complexity of an operation is constant, even if some operations may be expensive.

Let us consider an example of simple hash table insertions. How do we decide on table size? There is a trade-off between space and time, if we make hash-table size big, search time becomes low, but the space required becomes high.

Initially table is empty and size is 0

Insert Item 1 (Overflow)	1							
Insert Item 2 (Overflow)	1	2						
Insert Item 3	1	2	3					
Insert Item 4 (Overflow)	1	2	3	4				
Insert Item 5	1	2	3	4	5			
Insert Item 6	1	2	3	4	5	6		
Insert Item 7	1	2	3	4	5	6	7	

Next overflow would happen when we insert 9, table size would become 16

The idea is to increase the size of the table whenever it becomes full.
Following are the steps to follow when the table becomes full.

- 1) Allocate memory for larger table size, typically twice the old table.
- 2) Copy the contents of the old table to a new table.
- 3) Free the old table.

If the table has space available, we simply insert a new item in the available space.

What is the time complexity of n insertions using the above scheme?

If we use simple analysis, the worst-case cost of insertion is $O(n)$. Therefore, the worst-case cost of n inserts is $n * O(n)$ which is $O(n^2)$. This analysis gives an upper bound, but not a tight upper bound for n insertions as all insertions don't take $?(n)$ time.

Item No.	1	2	3	4	5	6	7	8	9	10
Table Size	1	2	4	4	8	8	8	8	16	16
Cost	1	2	3	1	5	1	1	1	9	1

$$\text{Amortized Cost} = \frac{(1 + 2 + 3 + 5 + 1 + 1 + 9 + 1\dots)}{n}$$

We can simplify the above series by breaking terms 2, 3, 5, 9.. into two as (1+1), (1+2), (1+4), (1+8)

$$\begin{aligned}\text{Amortized Cost} &= \frac{[(\underbrace{1 + 1 + 1 + 1\dots}_{n \text{ terms}}) + (\underbrace{1 + 2 + 4 + \dots}_{\lfloor \log_2(n-1) \rfloor + 1 \text{ terms}})]}{n} \\ &\leq \frac{[n + 2n]}{n} \\ &\leq 3\end{aligned}$$

$$\text{Amortized Cost} = O(1)$$

Potential Method

The potential approach focuses on how the current potential, may be calculated directly from the algorithm's or data structure's present state.

The potential technique chooses a function Φ that changes the data structure's states into non-negative values. S represents work that has been accounted for in the amortized analysis but has not yet been completed if S is taken to reflect the state of the data structure. Consequently, it is possible to think of $\Phi(S)$ as computing the quantity of potential energy that the state has to provide. The potential value is set to zero before a data structure is initialized. Alternately, it is possible to think of $\Phi(S)$ as denoting the degree of disorder or the separation of state S from the ideal state. We can designate a potential function Φ (read "Phi") on a data structure's states if it meets the criteria listed below.

1. $\Phi(a_0) = 0$, where a_0 is the starting state of the data structure.
2. $\Phi(a_t) \geq 0$ for all states a_t of the data structure occurring at the time of the course of the computation

At each stage in the computation, the potential function should be able to maintain track of the precharged time. It calculates the amount of time that can be saved up to cover expensive operations. Intriguingly, though, it simply depends on the data structure's current state, regardless of the history of the computation that led to that state.

We then define the amortized time of an operation as

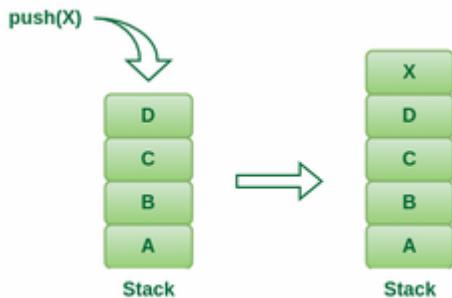
$c + \Phi(a') - \Phi(a)$, where c is the original cost of the operation and a and a' are the states of the data structure before and after the operation, respectively.

As a result, the amortized time is calculated as the actual time plus the prospective change. The amortized time of each operation should ideally be low when defined.

Analysis of potential method with example

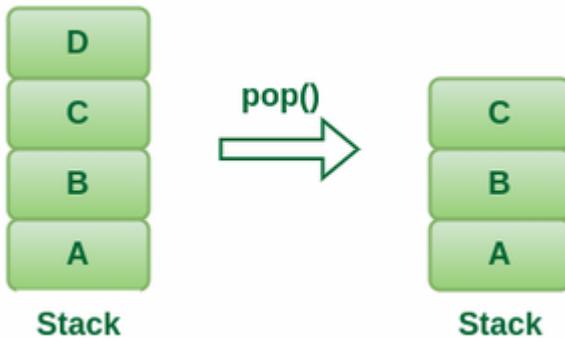
Stack operations

Push operation: Time complexity to push an item into a stack is $O(1)$



PUSH operation

Pop operation: Time complexity to pop an item from a stack is $O(1)$



POP Operation

Multipop operation: Time complexity to pop k items from a stack is $\min(\text{stack size}, k)$.

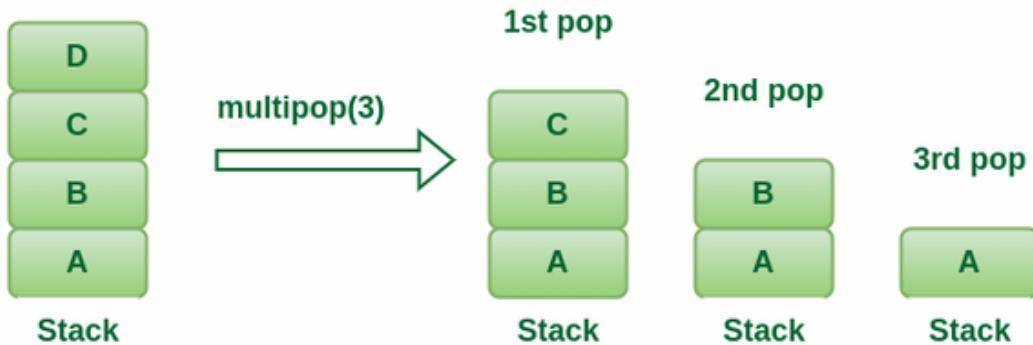
Algorithm:

Multipop (S, k)

While not Stack_Empty (S) and k != 0

do Pop (S)

k=k - 1



Accounting method (banker's)

The accounting method of amortized analysis can be useful for understanding the performance of algorithms that perform a sequence of operations with varying costs. It can be applied to a wide range of data structures and algorithms.

To solve a problem using amortized analysis using the accounting method, you can follow these steps:

- Identify the sequence of operations that the algorithm will perform, and determine which operations are “cheap” and which are “expensive.” Cheap operations are those that require less time or resources than the average operation, while expensive operations are those that require more time or resources.
- Define a credit or potential function that will be used to track the credit that has been accumulated by the algorithm. This function should assign a credit value to the state of the data structure at each point in time.
- Initialize the credit to 0.
- For each operation in the sequence, do the following:
 - If the operation is cheap, increment the credit by the cost of the operation.
 - If the operation is expensive, subtract the credit from the cost of the operation to determine the actual cost. The actual cost is the difference between the cost of the operation and the credit.
 - If the credit becomes negative, reset it to 0.
- At the end of the sequence, divide the total cost by the number of operations to determine the average cost per operation.

Here is an example of solving a problem using amortized analysis using the accounting method:

Suppose we have an algorithm that performs a series of insertions into a dynamic array. Each insertion is fast, but if the array becomes full, the

algorithm must perform a slower operation to resize the array and make room for the new insertion. We want to use amortized analysis to determine the average cost per insertion.

- Identify the sequence of operations: Each insertion is a cheap operation, and the resize operation is an expensive operation.
- Define a credit function: We can define the credit function as follows:
 - If the array is at least half empty, the credit is 0.
 - If the array is less than half empty, the credit is the number of empty slots in the array.
 - Initialize the credit to 0.
 - Perform the operations:

Insertion 1:

Cost = 1

Credit = 1

Insertion 2:

Cost = 1

Credit = 2

Insertion 3:

Array is full, must resize

Cost = 2 (insertion + resize)

Credit = 0

Insertion 4:

Cost = 1

Credit = 2

Insertion 5:

Cost = 1

Credit = 3

...

To calculate the average cost per operation, we sum the total cost and divide by the number of operations. In this example, the total cost is 8 and the number of operations is 5, so the average cost per operation is **8/5 = 1.6**.

This average cost per operation reflects the fact that some insertions have a lower cost due to the credit accumulated through previous cheap operations, while other insertions have a higher cost due to the expensive resize operation.

Disjoint Sets

- Disjoint sets are a collection of sets whose members do not overlap (no cycles) or are not duplicated in another set (mutually exclusive).
 - Pairwise Disjoint Sets

If S_i and S_j , $i \neq j$, are two sets, then there is no element that is in both S_i and S_j .
- Disjoint-set Data Structure is a data structure that keeps track of a set of elements partitioned into a number of disjoint sub-sets.
- **Disjoint Sets are Used**
 - To determine connected components in an undirected graph.
 - To find whether a graph contains cycle or not.

Disjoint Sets

- Majorly two important operations are done using disjoint sets
 - **Find** : Returns the subset containing the element e.
 - **Union** : Merges two disjoint sets.

Set Operations

- **Union**

If S_i and S_j are two disjoint sets, then their union $S_i \cup S_j = \{\text{all elements } x \text{ such that } x \text{ is in } S_i \text{ or } S_j\}$.

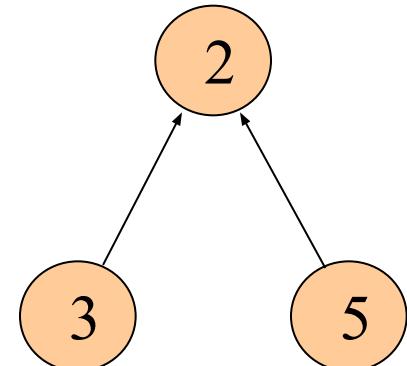
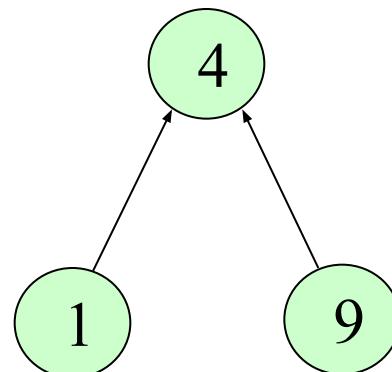
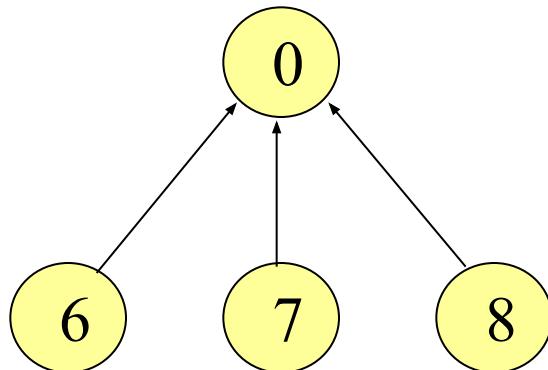
- **Find(i)**

Find the set containing element i.

Set Representation

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

$$n=10$$



$$S_1 = \{0, 6, 7, 8\}$$

$$S_2 = \{1, 4, 9\}$$

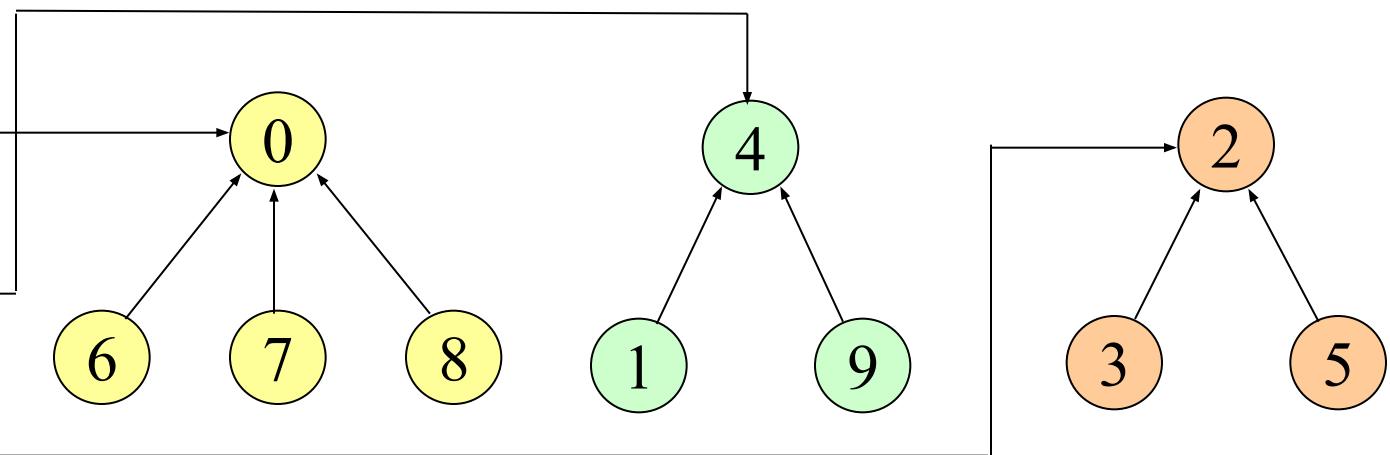
$$S_3 = \{2, 3, 5\}$$

Data Representation for S_1, S_2, S_3

Set

Name Pointer

S_1	
S_2	
S_3	

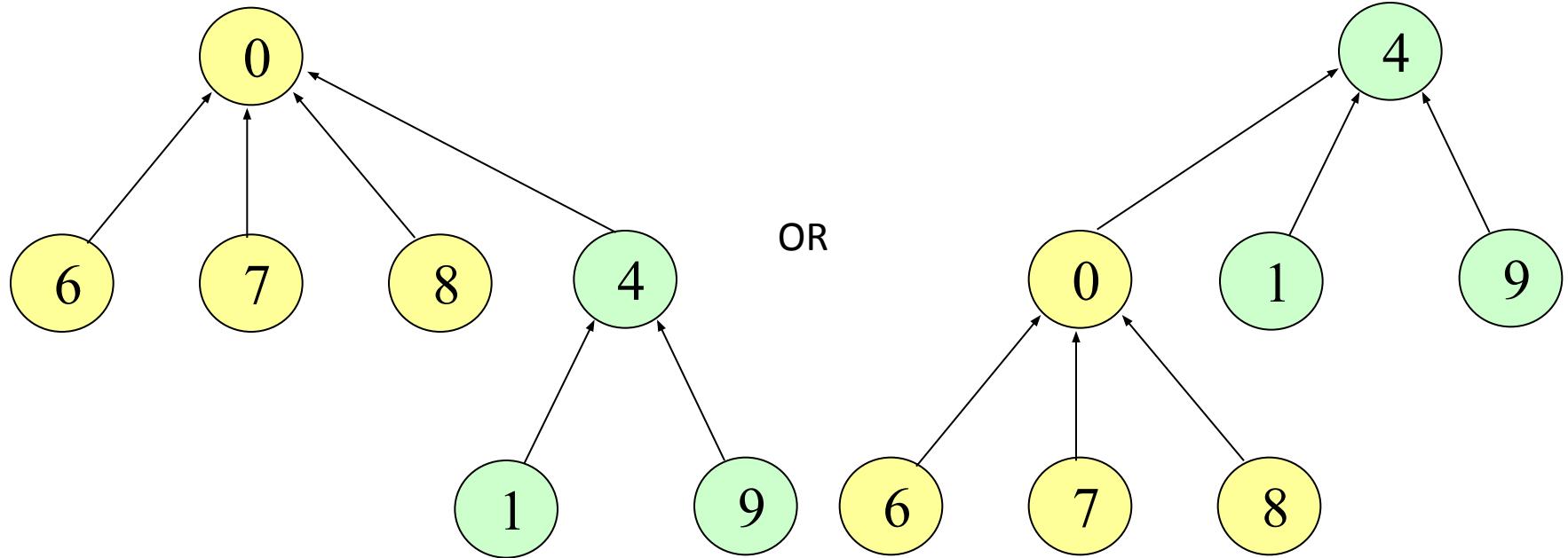


i	[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]
parent	-1	4	-1	2	-1	2	0	0	0	4

Array Representation of S_1, S_2, S_3

Disjoint Set Union

$S_1 \cup S_2$

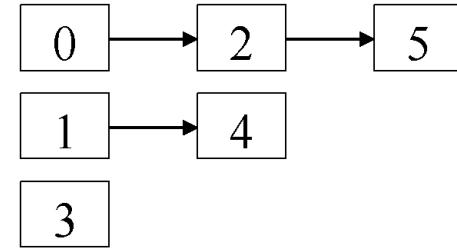


Disjoint Sets representation

- Disjoint sets are typically implemented using an array. Each item in disjoint set is represented with element and its representative.
- One of the element in the set is the representative.
- The '*i*' indicates the element and the '*p*' indicates representative element or another element in the set, giving rise to a linked list that eventually ends at the representative.

Note : $P[i]$ is -1 then *i* is the called the representative element of the set.

Six elements in three disjoint lists



Array of disjoint sets

P	-1	-1	0	-1	1	0
i	0	1	2	3	4	5

Algorithms Simple Union and Simple Find

- *SimpleUnion(i,j){*

```
Parent[i]=j; // making j as parent of i.  
}
```

- *Int simpleFind(i){*

```
while (Parent[i]>=0)  
    i=Parent[i]  
return i;  
}
```

Degenerate Tree

Union operation $O(n)$

Find operation $O(n^2)$



$\text{union}(0, 1), \text{find}(0)$

$\text{union}(1, 2), \text{find}(1)$

\vdots

$\text{union}(n-2, n-1), \text{find}(n-1)$

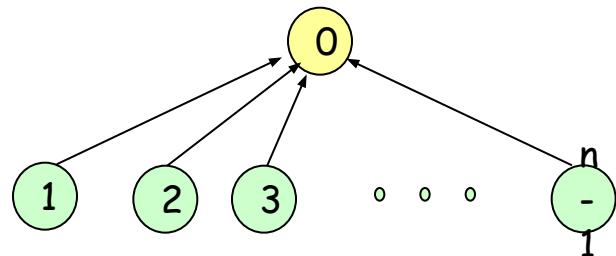
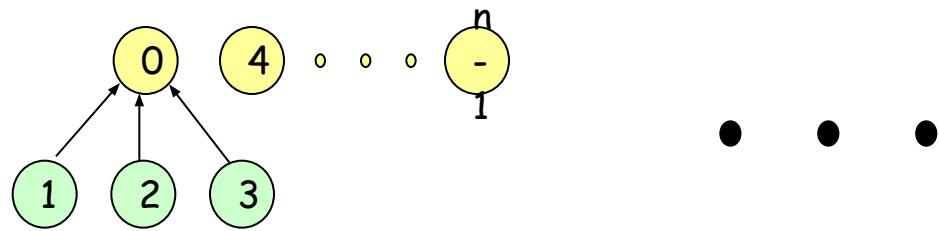
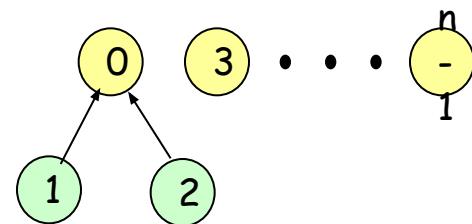
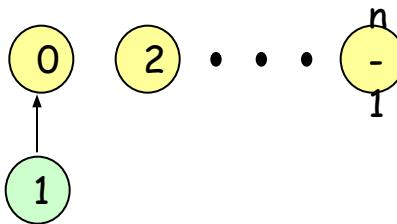
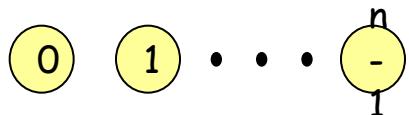
Time Complexity of Union and Find

- Each Union Operation takes constant time $O(1)$.
- For $n-1$ union operations the time required is $O(n)$ in total.
- Time required to find an element at level i of a tree is $O(i)$.
- Total time to process n find operations is $O(\sum_{i=1}^n i) = O(n^2)$.

Improve the performance of Set Union and Find Algorithms

- **Weighting Rule** [Weighting rule for $\text{union}(l, j)$]
 - If the number of nodes in the tree with root i is less than the number in the tree with root j , then make j the parent of i ; otherwise make i the parent of j .
- **Collapsing Rule**
 - If j is a node on the path from i to its root and $\text{parent}[i] \neq \text{root}(i)$, then set $\text{parent}[j]$ to $\text{root}(i)$.

Set Union with The Weighting Rule

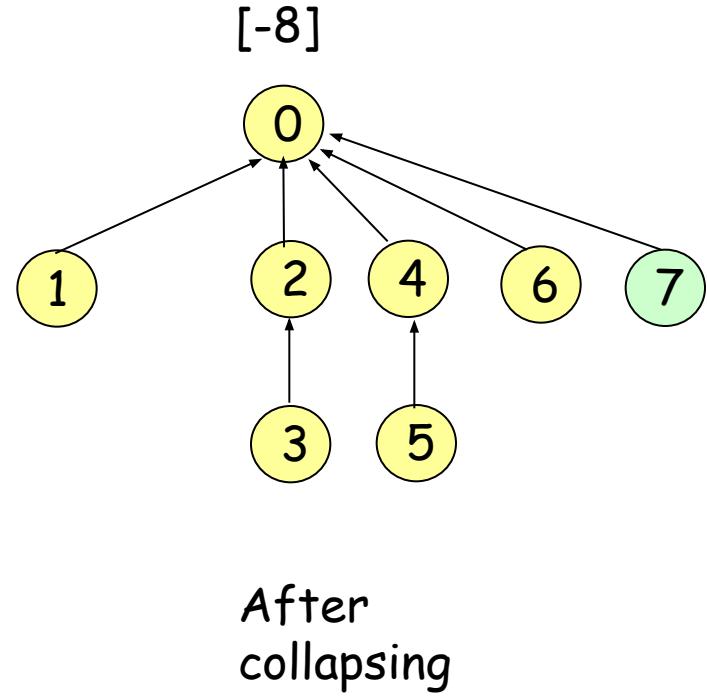
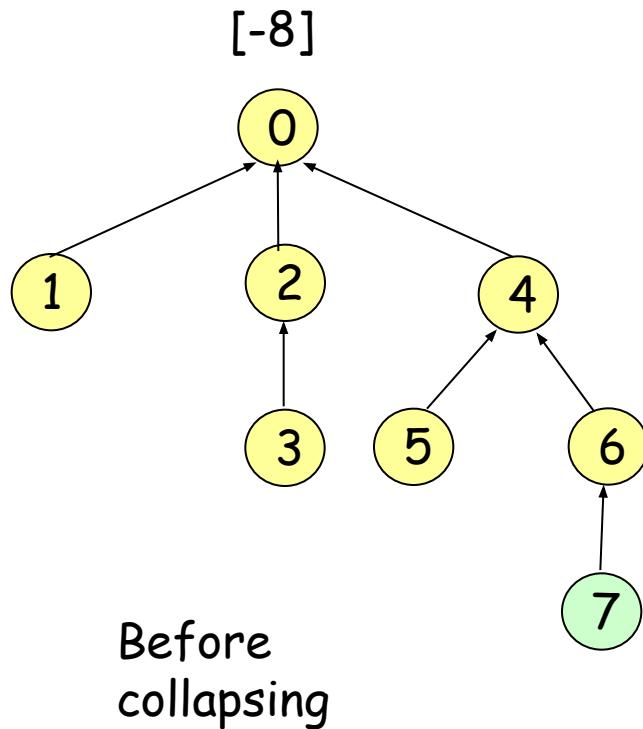


Weighted Union

- ***WeightedUnion(i,j)***

```
{  
    temp=Parent[i]+Parent [j];  
    if(Parent[i]>Parent [j])  
        {Parent[i]=j;  
         Parent [j]=temp;  
     }  
    else  
    {  Parent [j]=i;  
     Parent [i]=temp;  
    }  
}
```

Set Find with Collapsing Rule



CollapsingFind

- *CollapsingFind(i)*

```
{ r=i;  
  While(Parent[r]>0)  
    r=Parent[r];  
  while(i!=r)  
    {s=Parent[i];  
     Parent[i]=r;  
     i=s;  
    }  
  Return r;  
 }
```

DIVIDE AND CONQUER(DAC)

- **Divide and Conquer** is a Procedure which divides a problem into two or more sub-problems until these sub-problems become simple enough to be solved directly.
- The solutions of these sub-problems are combined to get the solution for the original problem.
- Often the sub-problems resulting from a divide and conquer design are of the same type as the original problem. Hence Divide and Conquer is a **recursive procedure**.
- **Control Abstraction**

It refers to a general procedure whose flow of control is clear but whose primary operations are specified by procedures which are not defined or they are different for different problems.

Divide-and-Conquer

- Divide-and conquer is a general algorithm design paradigm:
 - Divide: divide the input data S in two or more disjoint subsets S_1, S_2, \dots
 - Recur: solve the subproblems recursively
 - Conquer: combine the solutions for S_1, S_2, \dots , into a solution for S
- The base case for the recursion are subproblems of constant size
- Analysis can be done using recurrence equations

Control Abstraction

```
1  Algorithm DAndC( $P$ )
2  {
3      if Small( $P$ ) then return  $S(P)$ ;
4      else
5      {
6          divide  $P$  into smaller instances  $P_1, P_2, \dots, P_k$ ,  $k \geq 1$ ;
7          Apply DAndC to each of these subproblems;
8          return Combine(DAndC( $P_1$ ),DAndC( $P_2$ ),\dots,DAndC( $P_k$ ));
9      }
10 }
```

gorithm 3.1 Control abstraction for divide-and-conquer

DIVIDE AND CONQUER(DAC) (Contd..)

- **Analysis of DANDC** : The computing time of **DANDC** is described by a recurrence relation

$$T(n) = \begin{cases} T(1) & n=1 \\ aT(n/b) + f(n) & n>1 \end{cases} \quad \dots \text{ (1)}$$

Where

- $T(n)$ is the time for DandC on any input of size n
- $T(1)$ is the time to solve directly for small inputs.
- $f(n)$ is the time for dividing P and combining Sub-problems.
- a and b are known constants.
- We assume $T(1)$ is known and n is a power of b (ie $n=b^k$)

Recursive Linear Search

- Rec_Lnr_Srch(a, n, key)
 - {
 - if (n<1)
 - return -1
 - else
 - { if(a[n]==key)
 - return n
 - else
 - return Rec_Lnr_Srch(a, n--, key)
 - }
 - }

Solving Recurrence Relations

- Recurrence Relation
 - Given a recursive algorithm, a **recurrence relation for the algorithm** is an equation that gives the run time on an input size in terms of the run times of smaller input sizes.
- Solving Recurrence Relations
 - Iterative Method
 - Recurrence Tree Method
 - Master Method

Iterative Method

$$\begin{aligned} \text{e.g. } T(n) &= 2T(n/2) + n \\ &\Rightarrow 2[2T(n/4) + n/2] + n \\ &\Rightarrow 2^2T(n/4) + n + n \\ &\Rightarrow 2^2[2T(n/8) + n/4] + 2n \\ &\Rightarrow 2^3T(n/2^3) + 3n \end{aligned}$$

After k iterations , $T(n)=2kT(n/2^k)+kn$ -----(1)

Sub problem size is 1 after $n/2^k=1 \Rightarrow k=\log n$

So, after $\log n$ iterations ,the sub-problem size will be 1.

So, when $k=\log n$ is put in equation 1

$$\begin{aligned} T(n) &= nT(1) + n\log n \\ &= nc + n\log n \quad (\text{say } c=T(1)) \\ &= O(n\log n) \end{aligned}$$

Master Method

- The master method solves recurrences of the form
- $T(n) = aT(n/b) + f(n)$
- where $a \geq 1$ and $b > 1$ are constants and $f(n)$ is a asymptotically positive function .

$$T(n) = n^{\log_b a} [T(1) + u(n)]$$

- where $u(n) = \sum_{j=1}^k h(b^j)$ and $h(n) = f(n)/n^{\log_b a}$.

$h(n)$	$u(n)$
$O(n^r), r < 0$	$O(1)$
$\Theta((\log n)^i), i \geq 0$	$\Theta((\log n)^{i+1}/(i+1))$
$\Omega(n^r), r > 0$	$\Theta(h(n))$

- The table gives values of $T(n)$ for many recurrence relations encountered when analyzing div. and conq. problems

Solving Recurrence Relations

Beginning with the recurrence relation (1) and using substitution method it can be shown that

$$T(n) = n^{\log_b a} [T(1) + u(n)]$$

where $u(n) = \sum_{j=1}^k h(b^j)$ and $h(n) = f(n)/n^{\log_b a}$.

The following table tabulates asymptotic values of $u(n)$ for various values of $h(n)$

$h(n)$	$u(n)$
$O(n^r), r < 0$	$O(1)$
$\Theta((\log n)^i), i \geq 0$	$\Theta((\log n)^{i+1}/(i+1))$
$\Omega(n^r), r > 0$	$\Theta(h(n))$

The table allows one to easily obtain the asymptotic values of $T(n)$ for many recurrence relations encountered when analyzing div. and conq. problems

BINARY SEARCH (Contd..)

Algorithm BinSrch(a, i, l, x)

```
// Given an array  $a[i : l]$  of elements in nondecreasing
// order,  $1 \leq i \leq l$ , determine whether  $x$  is present, and
// if so, return  $j$  such that  $x = a[j]$ ; else return 0.
{
    if ( $l = i$ ) then // If Small( $P$ )
    {
        if ( $x = a[i]$ ) then return  $i$ ;
        else return 0;
    }
    else
    { // Reduce  $P$  into a smaller subproblem.
         $mid := \lfloor (i + l)/2 \rfloor$ ;
        if ( $x = a[mid]$ ) then return  $mid$ ;
        else if ( $x < a[mid]$ ) then
            return BinSrch( $a, i, mid - 1, x$ );
        else return BinSrch( $a, mid + 1, l, x$ );
    }
}
```

BINARY SEARCH (Contd..)

Example: n = 9

A(j:n) are -15, -6, 0, 7, 9, 23, 54, 82, 101. x is 9, 101, -14, 82

Case 1

x=101

i l mid

1 9 5

6 9 7

8 9 8

9 9 9

found at 9

Case 2

x= -14

i l mid

1 9 5

1 4 2

1 1 1

2 1

not found

Case3

x=82

i l mid

1 9 5

6 9 7

5 9 8

found at 8

1. Draw the binary decision tree for the following set

-15, -6, 0, 7, 9, 23, 54, 82, 101.

2. Draw the binary decision tree for the following set

(3, 6, 9, 12, 15, 18, 21, 24, 27, 30, 33, 36, 39, 42, 45, 47)

BINARY SEARCH (Contd..)

- The Binary decision tree is a tree with value of mid index at the nodes.
- If x is present the algorithm will end at one of circular nodes or internal nodes; otherwise at square nodes or external nodes.
- The worst case time for binary search with n elements (n in $[2^{k-1}, 2^k]$ for some integer k) is
 - $O(\log n)$ for successful search
 - $\Theta(\log n)$ for unsuccessful search.
 - $O(\log n)$ – lower limit and
 - $\Theta(\log n)$ – Lower and upper limit

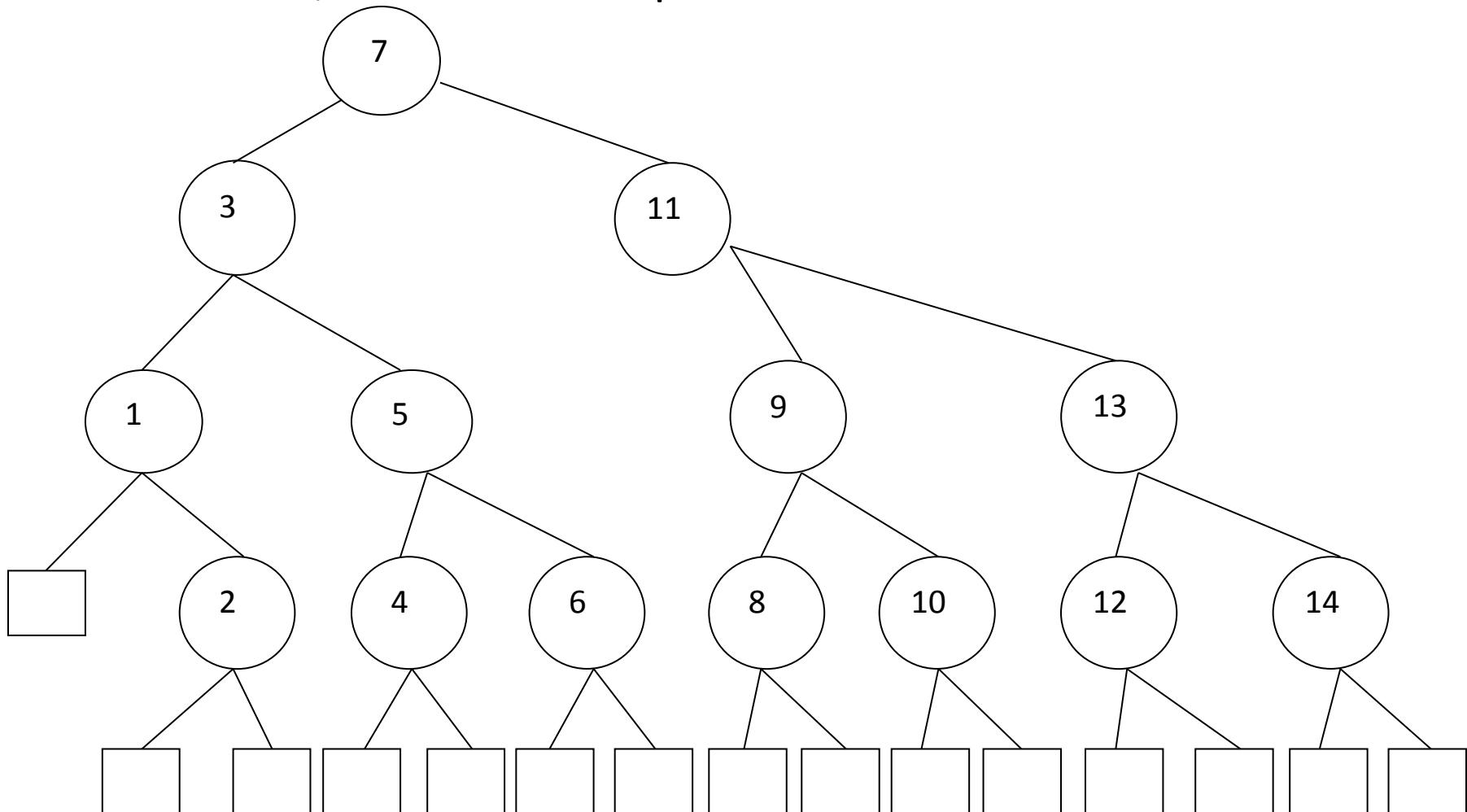
For $n=14$ it looks as below

1 – 14 mid 7

Binary Decision Tree for Binary Search (Contd..)

The Binary decision tree is a tree with **value of mid index** at the nodes.

If x is present the algorithm will end at one of **circular nodes** or internal nodes; otherwise at square nodes or **external nodes**.



BINARY SEARCH (Contd..)

So the recurrence relation is

$$T(n) = \begin{cases} c & \text{when } n \text{ is small or} \\ & \text{search element is first mid element} \\ T(n/2) + 3 & \text{atleast 3 comparisions in worst case} \\ & \text{before recursive binary search is invoked} \end{cases}$$

Solving T(n)

$$T(n) = T(n/2) + 3$$

$$T(n/2) = T(n/4) + 3 + 3$$

Substitute $T(n/2)$ value in

$$T(n) = T(n/4) + 3 + 3$$

... Repeating k times - $T(n/2^k) + 3 + 3 + \dots + 3$ k times

$$\text{Let } 2^k = n \Rightarrow k = \log n$$

$$T(n) = T(1) + k = c + \log n \in O(\log n)$$

Successful Searches

Best average worst

$$\theta(1) \quad \theta(\log n) \quad O(\log n)$$

Unsuccessful Searches

best average worst

$$\theta(\log n)$$

Merge Sort

Given a sequence of n elements $a[1], \dots, a[n]$, the idea is to imagine them split into two sets $a[1], a[2], \dots, a[n/2]$ and $a[n/2 + 1], \dots, a[n]$. Each set is individually sorted and the resulting sorted sequences are merged to produce a single sorted sequence of n elements.

In General

Merge-sort on an input sequence S with n elements consists of three steps:

- **Divide:** partition S into two sequences S_1 and S_2 of about $n/2$ elements each
- **Recur:** recursively sort S_1 and S_2
- **Conquer:** merge S_1 and S_2 into a unique sorted sequence

```
MergeSort(low, high){  
    if(low < high) then{  
        // Divide p into subproblems.  
        Mid := (low + high )/2;  
        // Solve the subproblems  
        MergeSort(low, mid);  
        MergeSort(mid+1, high);  
        // Combine the solutions  
        Merge(low, mid, high);  
    }  
}
```

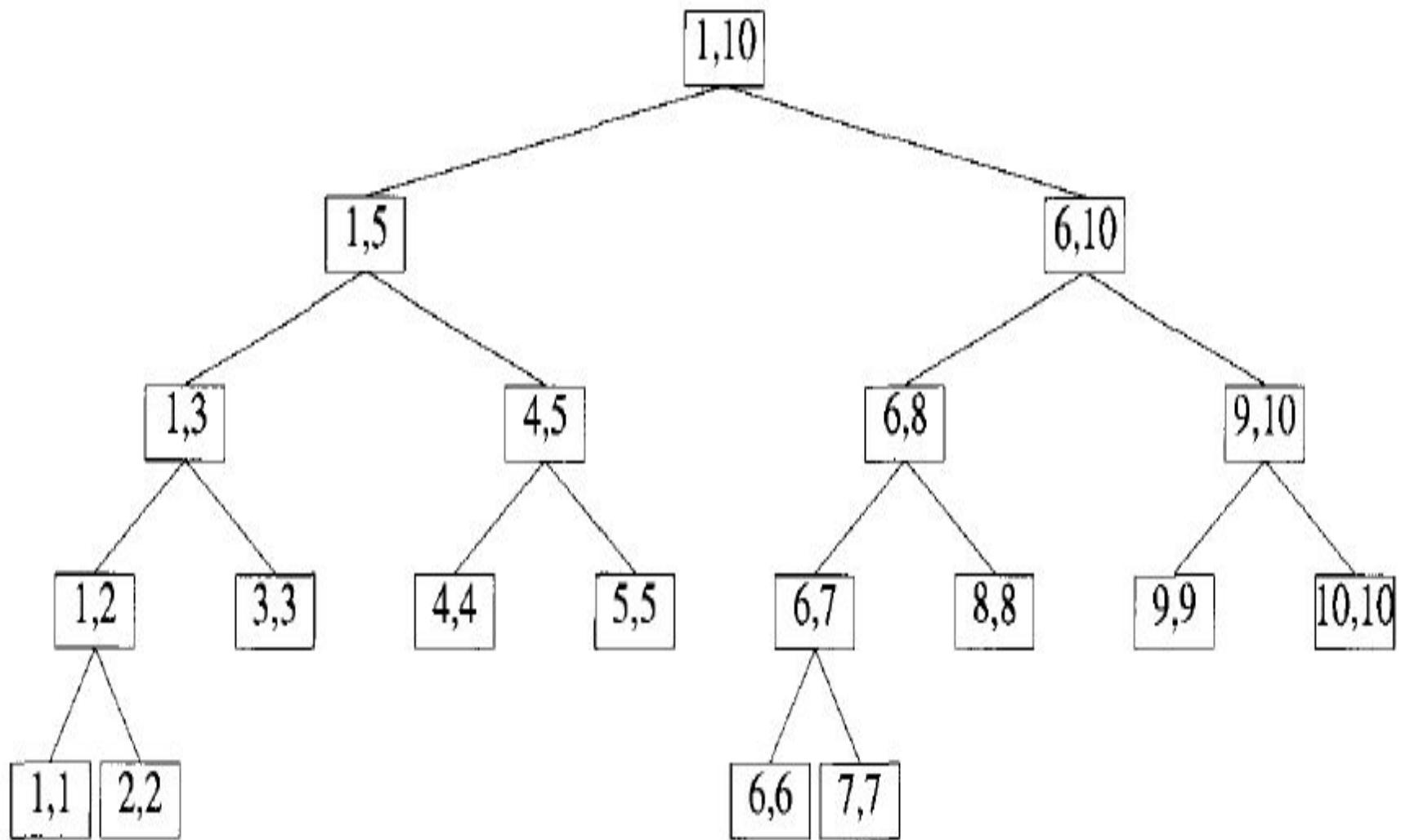
Merge

Merge(low, mid, high)

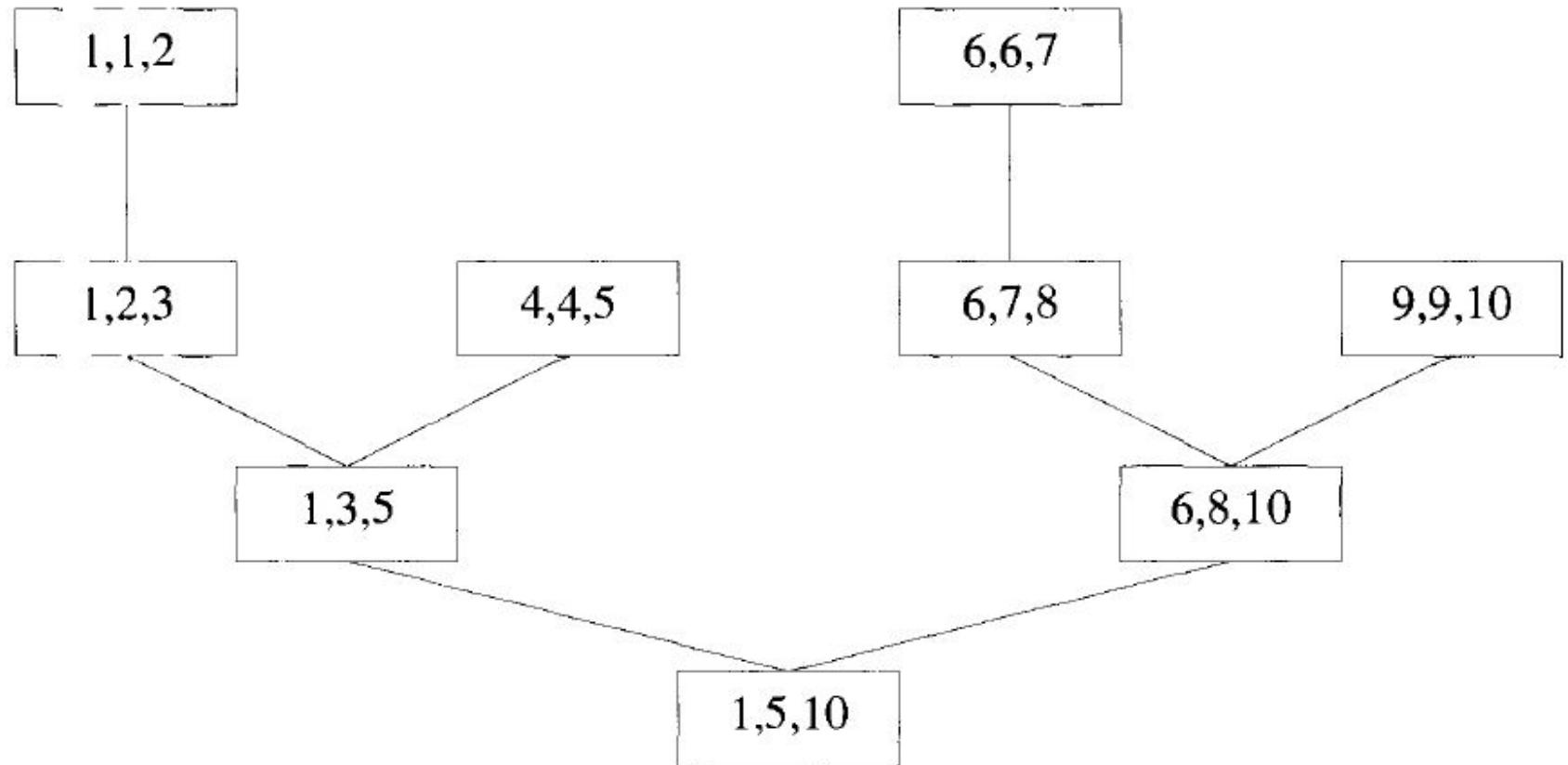
```
{  
    i=h=low; j :=mid+1;  
    while((h<=mid) and (j<= high))  
        do  
        {  
            if(a[i] <= a[j]) then  
            { b[h] :=a[i];  
                i := i+1  
                h := h + 1;  
            }  
            else  
            {  
                b[h] :=a[j];  
                j := j +1;}  
                h := h +1;  
        }  
}
```

```
if(h>mid) then  
    for k=j to high do  
    { b[h]=a[k];  
        h := h +1;  
    }  
    else  
        For k :=i to mid do  
        {  
            b[h] :=a[k];  
            h := h +1;  
        }  
    for k := low to high do  
        a[k] := b[k];  
    }
```

Tree of calls of Mergesort(1,10)



Tree of calls of Merge



Merge Sort Analysis

- The conquer step of merge-sort consists of merging two sorted sequences, each with $n/2$ elements, takes at most bn steps, for some constant b .
- Likewise, the basis case ($n < 2$) will take at b most steps.
- Therefore, if we let $T(n)$ denote the running time of merge-sort:

$$T(n) = \begin{cases} b & \text{if } n < 2 \\ 2T(n/2) + bn & \text{if } n \geq 2 \end{cases}$$

- We can therefore analyze the running time of merge-sort by finding a **closed form solution** to the above equation.
 - That is, a solution that has $T(n)$ only on the left-hand side.

Iterative Substitution

- In the iterative substitution, or “plug-and-chug,” technique, we iteratively apply the recurrence equation to itself and see if we can find a pattern:

$$\begin{aligned}T(n) &= 2T(n/2) + bn \\&= 2(2T(n/2^2)) + b(n/2) + bn \\&= 2^2 T(n/2^2) + 2bn \\&= 2^3 T(n/2^3) + 3bn \\&= 2^4 T(n/2^4) + 4bn \\&= \dots \\&= 2^i T(n/2^i) + ibn\end{aligned}$$

- Note that base, $T(n)=b$, case occurs when $2^i=n$. That is, $i = \log n$.
- So,
- Thus, $T(n)$ is $O(n \log n)$.

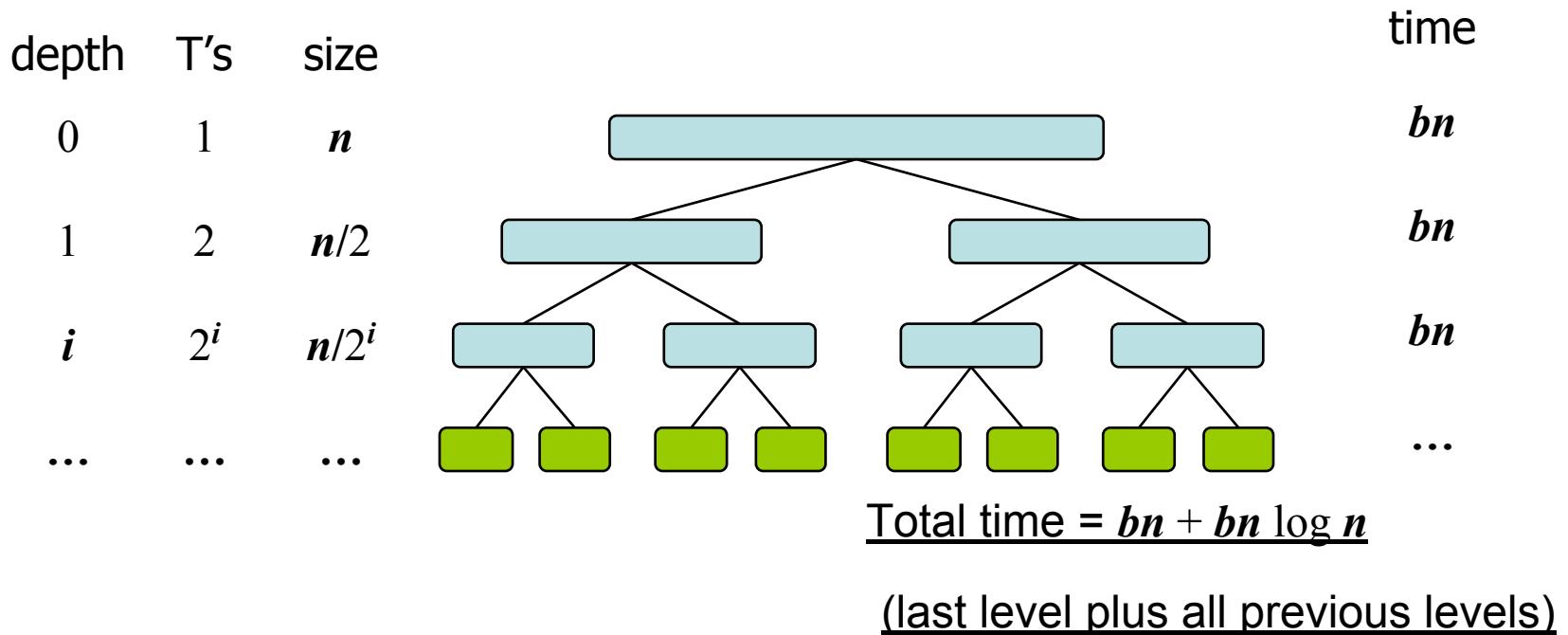
$$T(n) = bn + bn \log n$$

The Recursion Tree



- Draw the recursion tree for the recurrence relation and look for a pattern:

$$T(n) = \begin{cases} b & \text{if } n < 2 \\ 2T(n/2) + bn & \text{if } n \geq 2 \end{cases}$$



- Draw the tree of calls of merge sort for the following set.

(35, 25, 15, 10, 45, 75, 85, 65, 55, 5, 20, 18)

Quicksort

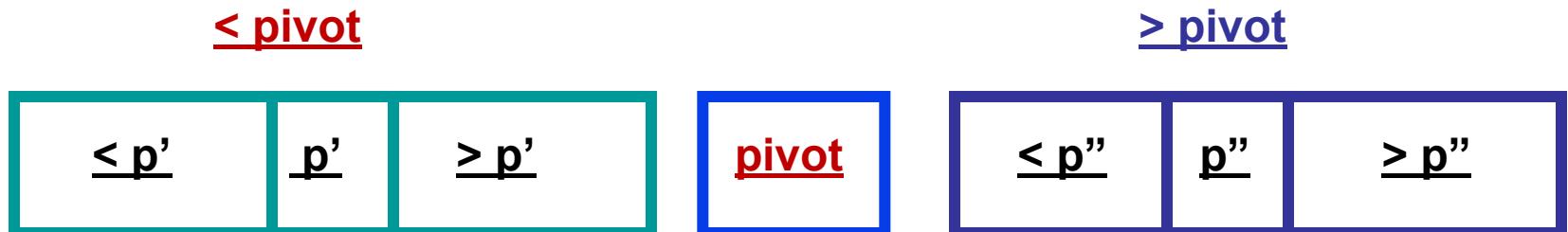
- In Quick sort, the division into 2 sub arrays is made so that the sorted sub arrays **need not be merged**.
- **Given a sequence of n elements $a[1], \dots, a[n]$, the idea is to rearrange the elements in $a[1:n]$ such that $a[i] \leq a[j]$ for all i , between $i & m$ and $\forall j$, between $m+1 & n$, for some m , $1 \leq m \leq n$.**
- Thus, the elements in $a[1:m]$ and $a[m+1 : n]$ can be independently sorted.
- This rearranging is referred to as **Partitioning**
- Efficient sorting algorithm - Discovered by C.A.R. Hoare
- Example of **Divide and Conquer** algorithm
- Two phases
 - Partition phase : **Divides the work into half**
 - Sort phase : **Conquers the halves!**

Quicksort

- Partition
 - Choose a **pivot** and rearrange the element around pivot
 - Find the position for the pivot so that
 - all elements to the left are less than pivot
 - all elements to the right are greater than pivot



- Conquer
 - Apply the same algorithm to each half



Quicksort - Partition

```
int partition( int *a, int low, int high ) {  
    int left, right;  
    int pivot_item;  
    pivot_item = a[low];  
    pivot = left = low;  
    right = high;  
    while ( left < right ) {  
        /* Move left while item <= pivot */  
        while( a[left] <= pivot_item ) left++;  
        /* Move right while item > pivot */  
        while( a[right] >= pivot_item ) right--;  
        if ( left < right ) SWAP(a,left,right);  
    }  
    /* right is at its final position */  
    a[low] = a[right];  
    a[right] = pivot_item;  
    return right;  
}
```

This example uses int's to keep things simple!

Any item will do as the pivot, choose the leftmost one!



high

Quicksort - Partition

```
int partition( int *a, int low, int high ) {  
    int left, right;  
    int pivot_item;  
    pivot_item = a[low];  
    pivot = left = low;  
    right = high;  
    while ( left < right ) {  
        /* Move left while item < pivot */  
        while left a[left] <= pivot_item ) left++ right  
        /* right while item > pivot */  
        while( a[right] >= pivot_item ) right--  
        if  
    }  
    /* right is final position for the pivot */  
    a[low] [right]; pivot: 23  
    a[rig : pivot_i  
    return right;  
}
```

Set left and right markers

low

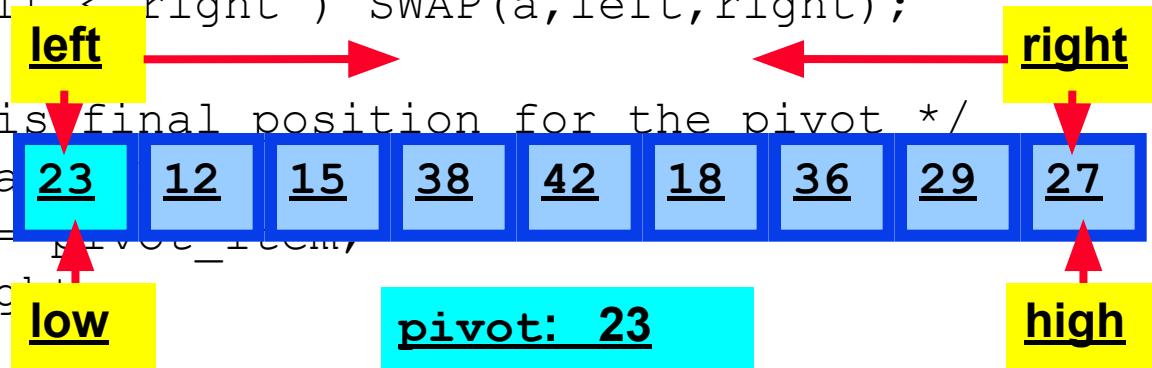
high

pivot: 23

Quicksort - Partition

```
int partition( int *a, int low, int high ) {  
    int left, right;  
    int pivot_item;  
    pivot_item = a[low];  
    pivot = left = low;  
    right = high;  
    while ( left < right ) {  
        /* Move left while item < pivot */  
        while( a[left] <= pivot_item ) left++;  
        /* Move right while item > pivot */  
        while( a[right] >= pivot_item ) right--;  
        if ( left < right ) SWAP(a, left, right);  
    }  
    /* right is final position for the pivot */  
    a[low] = a[pivot_item];  
    a[right] = pivot_item;  
    return right;  
}
```

Move the markers
until they cross over



Quicksort - Partition

```
int partition( int *a, int low, int high ) {  
    int left, right;  
    int pivot_item;  
    pivot_item = a[low];  
    pivot = left = low;  
    right = high;  
    while ( left < right ) {  
        /* Move left while item < pivot */  
        while( a[left] <= pivot_item ) left++;  
        /* Move right while item > pivot */  
        while( a[right] >= pivot_item ) right--;  
        if ( left < right ) SWAP(a, left, right);  
    }  
    /* right is final position for the pivot */  
    return right;  
}
```

Move the left pointer while it points to items \leq pivot

Move right similarly



return right;

low

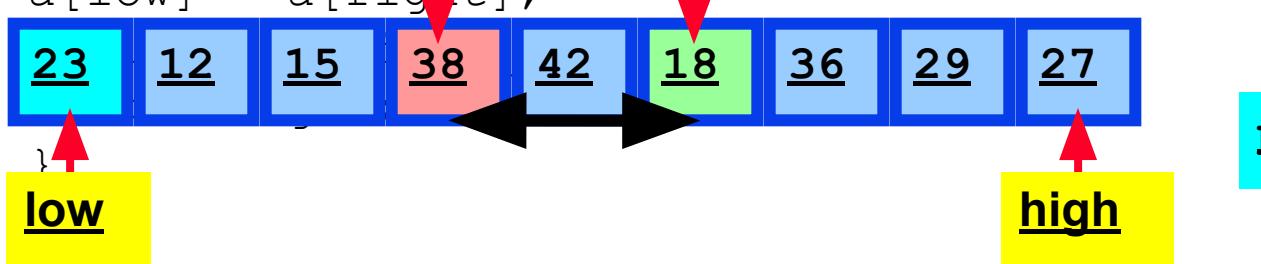
pivot: 38

high

Quicksort - Partition

```
int partition( int *a, int low, int high ) {  
    int left, right;  
    int pivot_item;  
    pivot_item = a[low];  
    pivot = left = low;  
    right = high;  
while ( left < right ) {  
    /* Move left while item < pivot */  
    while( a[left] <= pivot_item ) left++;  
    /* Move right while item > pivot */  
    while( a[right] >= pivot_item ) right--;  
    if ( left < right ) SWAP(a, left, right);  
}  
/* right is left pos right for the pivot */  
a[low] = a[right];
```

Swap the two items
on the wrong side of the pivot



pivot: 23

Quicksort - Partition

```
int partition( int *a, int low, int high ) {  
    int left, right;  
    int pivot_item;  
    pivot_item = a[low];  
    pivot = left = low;  
    right = high;  
    while ( left < right ) {  
        /* Move left while item < pivot */  
                while( a[left] <= pivot_item ) left++;  
        /* Move right while item > pivot */  
                while( a[right] >= pivot_item ) right--;  
        if ( left < right ) SWAP(a, left, right);  
    }  
    /* right is final position for the pivot */  
    a[low] = a[right];  
}
```

left and right
have swapped over,
so stop



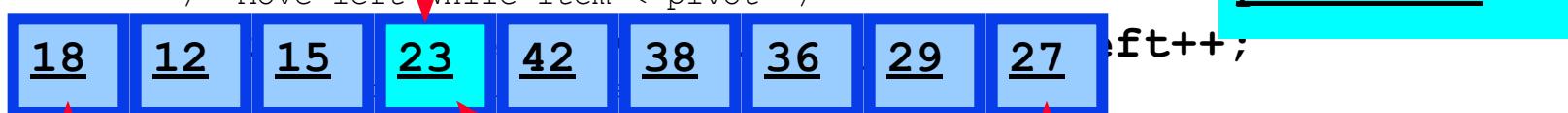
Quicksort - Partition

```
int partition( int *a, int low, int high ) {  
    int left, right;  
    int pivot_item;  
    pivot_item = a[low];  
    pivot = left = low;  
    right = high; // right is index of element to swap  
    while ( left < right ) {  
        /* Move left while item < pivot */  
        while( a[left] < pivot_item ) left++;  
        while( a[right] >= pivot_item ) right--;  
        if ( left < right ) SWAP(a, left, right);  
    }  
    /* right is final position for the pivot */  
    a[low] = a[right];  
    a[right] = pivot_item;  
    return right;  
}
```

Finally, swap the pivot
and right

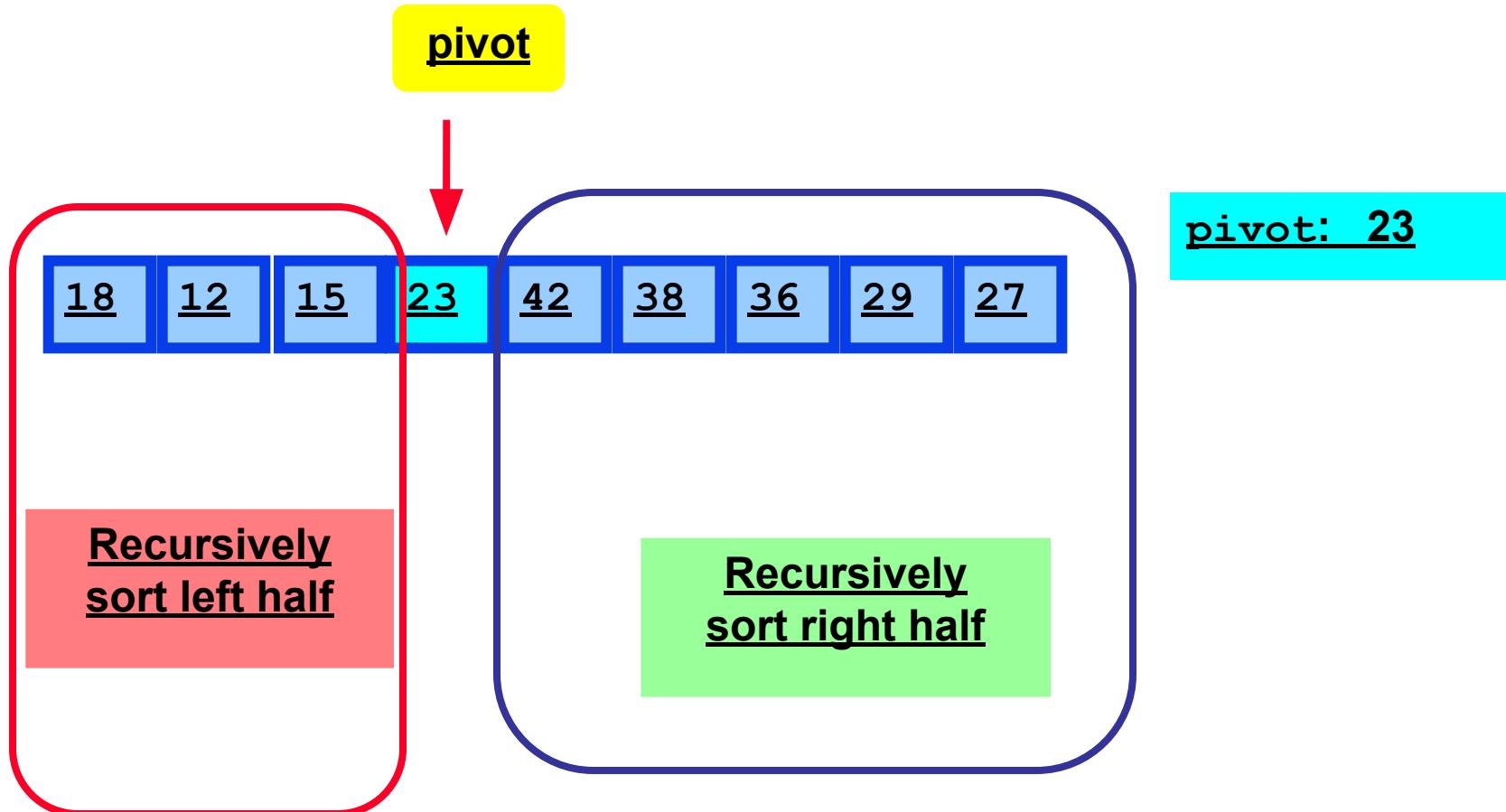
Quicksort - Partition

```
int partition( int *a, int low, int high ) {  
    int left, right;  
    int pivot_item;  
    pivot_item = a[low];  
    pivot = left = low;  
    right = hi right  
    while ( low < right ) {  
        /* Move left while item < pivot */  
        if ( a[left] < pivot_item ) left++;  
        else  
            while( a[right] >= pivot_item ) right--;  
        if ( left < right ) SWAP(a, l high ght);  
    }  
    /* right is final position for the pivot */  
    a[low] = a[right];  
    a[right] = pivot_item;  
    return right;  
}
```



**Return the position
of the pivot**

Quicksort - Conquer



Quick Sort

Algorithm QuickSort(p,q)

{

// Sorts the elements a[p]...,a[q] which reside in global array
// a[1:n] into ascending order.

if (p<q) then

{// Divide P into two subproblems.

j:= Partition(a, p, q)

// Solve the subproblems

Quicksort(p,j-1);

Quicksort(j+1,q);

//There is no need for combining solutions.

}

}

Quicksort - Partition

```
int partition( int *a, int low, int high )
{
    int left, right;
    pivot = left = low;
    right = high;
    while ( left < right )
    {
        /* Move left while item < pivot */
        while( a[left] <= a[pivot])
            left++;

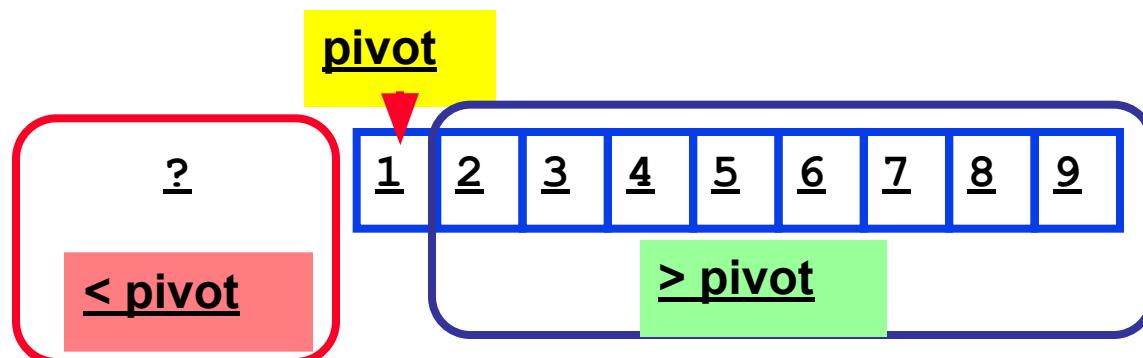
        /* Move right while item > pivot */
        while( a[right] >= a[pivot])
            right--;

        if ( left < right )
            SWAP(a, left, right);
    }
}
```

```
/* right >= left */
temp = a[pivot];
a[pivot] = a[right];
a[right] = temp;
return right;
```

Time Complexity

- What happens if we use quicksort on data that's already sorted (*or nearly sorted*)
- *We'd certainly expect it to perform well!*



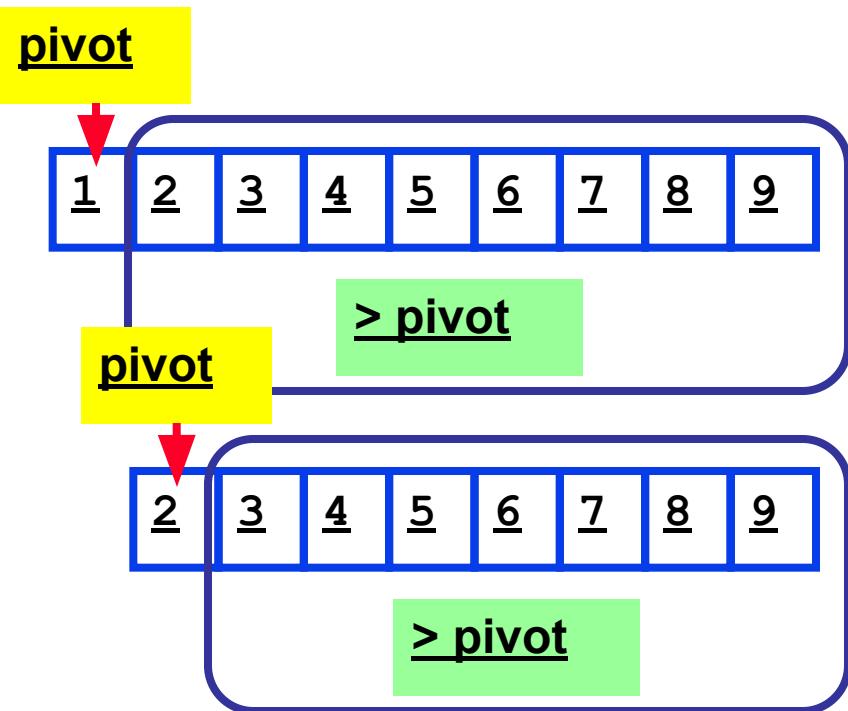
Quicksort - The truth!

- Sorted data
- Each partition produces
 - a problem of size 0
 - and one of size $n-1$!
- Number of partitions?

n each needing time $O(n)$

Total $nO(n)$
or $O(n^2)$

Quicksort is as bad as bubble or insertion sort



Quicksort - The truth!

- Quicksort's $O(n \log n)$ behavior
 - Depends on the partitions being nearly equal
 - there are $O(\log n)$ of them
- On average, this will *nearly* be the case **and** quicksort is generally $O(n \log n)$
- Can we do anything to ensure $O(n \log n)$ time?
- In general, no
 - But we can improve our chances!!

Average case Time complexity

Let the partitioning element $pivot$ has the equal probability of being the j th smallest element, in $a[m:p]$ elements. Hence two sub arrays remaining to be sorted are $a[m:j]$ and $a[j+1:p]$ with probability $1/(p-m)$, $m \leq j \leq p$. From this we obtain the recurrence relation

The no. of elements comparisons required by Partition on its first call is $n+1$. Note $T(0) = T(1) = 0$. multiply both sides of (1) by n , we obtain

Replacing n by $n-1$ in (2) gives

Subtracting (3) from (2)

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

Average case Time complexity

Repeatedly using to substitute for $T(n-1)$, $T(n-2)$ We get

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

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

.....

.....

$$= T(1)/2 + 2 \sum_{3 \leq k \leq n+1} 1/k$$

$$= 2 \sum_{3 \leq k \leq n+1} 1/k$$

since

$$\sum_{3 \leq k \leq n+1} 1/k \leq \int_2^{n+1} 1/x \, dx = \log(n+1) - \log 2 = \log n$$

$$T(n) = (n+1) * \log n$$

$$= n \log n + \log n$$

$$= \Theta(n \log n)$$

Basic Matrix Multiplication

Let A and B be two matrices of size $n \times n$, the product matrix C is computed as $C=AB$, is also an $n \times n$ matrix whose i^{th} , j^{th} element is formed by taking the elements in the i^{th} row of A and the j^{th} column of B.

```
void matrix_mult (){  
    for (i = 1; i <= N; i++) {  
        for (j = 1; j <= N; j++) {  
            for(k=1; k<=N; k++)  
                Ci,j = Ci,j + Ai,k * Bk,j  
        }  
    }  
}
```

Time_analysis

$$C_{i,j} = \sum_{k=1}^N a_{i,k} b_{k,j}$$

$$\text{Thus } T(N) = \sum_{i=1}^N \sum_{j=1}^N \sum_{k=1}^N c = cN^3 = O(N^3)$$

To compute $C_{i,j}$ N multiplications are required

The matrix has N^2 elements and hence it requires N^3 time

Basic Matrix Multiplication

Suppose we want to multiply two matrices of size $N \times N$:
for example $A \times B = C$.

$$C_{11} = A_{11}B_{11} + A_{12}B_{21}$$

$$C_{12} = A_{11}B_{12} + A_{12}B_{22}$$

$$C_{21} = A_{21}B_{11} + A_{22}B_{21}$$

$$\mathbf{C}_{22} = \mathbf{A}_{21}\mathbf{B}_{12} + \mathbf{A}_{22}\mathbf{B}_{22}$$

2x2 matrix multiplication can be accomplished in 8 multiplication.
 $(2^{\log_2 8} = 2^3)$

- In each of the 4 equations we have two multiplications of $n/2 \times n/2$ matrices that are then added together.
 - A recursive, divide-and-conquer algorithm can then be devised

Matrix Multiplication using Div & Conq

- Let us first assume that n is an exact power of 2 in each of the n x n matrices for A and B.
- Now break n x n matrix into smaller blocks or quadrants of size n/2 x n/2, This process is termed as **block partitioning**.
- Once matrices are split into blocks and multiplied, the blocks behave as if they were atomic elements.
- Consider two matrices A and B with 4x4 dimension each as shown below

$$\begin{bmatrix} c_{11} & c_{12} & c_{13} & c_{14} \\ c_{21} & c_{22} & c_{23} & c_{24} \\ c_{31} & c_{32} & c_{33} & c_{34} \\ c_{41} & c_{42} & c_{43} & c_{44} \end{bmatrix} = \begin{bmatrix} a_{11} & a_{12} & a_{13} & a_{14} \\ a_{21} & a_{22} & a_{23} & a_{24} \\ a_{31} & a_{32} & a_{33} & a_{34} \\ a_{41} & a_{42} & a_{43} & a_{44} \end{bmatrix} \begin{bmatrix} b_{11} & b_{12} & b_{13} & b_{14} \\ b_{21} & b_{22} & b_{23} & b_{24} \\ b_{31} & b_{32} & b_{33} & b_{34} \\ b_{41} & b_{42} & b_{43} & b_{44} \end{bmatrix}$$

$$c_{11} = a_{11} * b_{11} + a_{12} * b_{21} + a_{13} * b_{31} + a_{14} * b_{41} \quad (1)$$

$$c_{12} = a_{11} * b_{12} + a_{12} * b_{22} + a_{13} * b_{32} + a_{14} * b_{42} \quad (2)$$

$$c_{21} = a_{21} * b_{11} + a_{22} * b_{21} + a_{23} * b_{31} + a_{24} * b_{41} \quad (3)$$

$$c_{22} = a_{21} * b_{12} + a_{22} * b_{22} + a_{23} * b_{32} + a_{24} * b_{42} \quad (4)$$

Matrix Multiplication

$$\begin{array}{c} \textbf{A} \\ \left[\begin{array}{cc|cc} a_{11} & a_{12} & a_{13} & a_{14} \\ a_{21} & a_{22} & a_{23} & a_{24} \end{array} \right] \\ \left[\begin{array}{cccc} a_{31} & a_{32} & a_{33} & a_{34} \\ a_{41} & a_{42} & a_{43} & a_{44} \end{array} \right] \end{array} \quad \begin{array}{c} \textbf{B} \\ \left[\begin{array}{cc|cc} b_{11} & b_{12} & b_{13} & b_{14} \\ b_{21} & b_{22} & b_{23} & b_{24} \end{array} \right] \\ \left[\begin{array}{cccc} b_{31} & b_{32} & b_{33} & b_{34} \\ b_{41} & b_{42} & b_{43} & b_{44} \end{array} \right] \end{array}$$

$$A_{11} * B_{11} + A_{12} * B_{21} = \left[\begin{array}{cccc} c_{11} & c_{12} & \cdot & \cdot \\ c_{21} & c_{22} & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot \end{array} \right]$$

So the idea is to recursively divide $n \times n$ matrices into $n/2 \times n/2$ matrices until they are small enough to be multiplied in the naive way

Matrix Multiplication with div & conq

- Divide matrices into sub-matrices: A_0, A_1, A_2 etc
- Use blocked matrix multiply equations
- Recursively multiply sub-matrices
- For multiplying two matrices of size $n \times n$, we make 8 recursive calls above, each on a matrix/sub-problem with size $n/2 \times n/2$.
- Each of these recursive calls multiplies two $n/2 \times n/2$ matrices, which are then added together.

$$T(n) = \begin{cases} b & n \leq 2 \\ 8T(n/2) + cn^2 & n > 2 \end{cases}$$

Hence $T(n) = O(n^3)$

Strassens Matrix Multiplication

- Strassen's algorithm makes use of the same divide and conquer approach as above but is asymptotically faster, i.e. $O(n \lg 7)$.
- The usual multiplication of two 2×2 matrices takes 8 multiplications and 4 additions. Strassen showed how two 2×2 matrices can be multiplied using only 7 multiplications and 18 additions or subtractions

Strassen Matrix Multiplication

- Strassen showed that 2x2 matrix multiplication can be accomplished in 7 multiplication of $n/2 \times n/2$ and 18 additions or subtractions.
 $(2^{\log_2 7} = 2^{2.807})$
- This reduce can be done by Divide and Conquer Approach.

$$\begin{vmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{vmatrix} = \begin{vmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{vmatrix} \begin{vmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{vmatrix}$$

$$P_1 = (A_{11} + A_{22})(B_{11} + B_{22})$$

$$P_2 = (A_{21} + A_{22}) * B_{11}$$

$$P_3 = A_{11} * (B_{12} - B_{22})$$

$$P_4 = A_{22} * (B_{21} - B_{11})$$

$$P_5 = (A_{11} + A_{12}) * B_{22}$$

$$P_6 = (A_{21} - A_{11}) * (B_{11} + B_{12})$$

$$P_7 = (A_{12} - A_{22}) * (B_{21} + B_{22})$$

$$C_{11} = P_1 + P_4 - P_5 + P_7$$

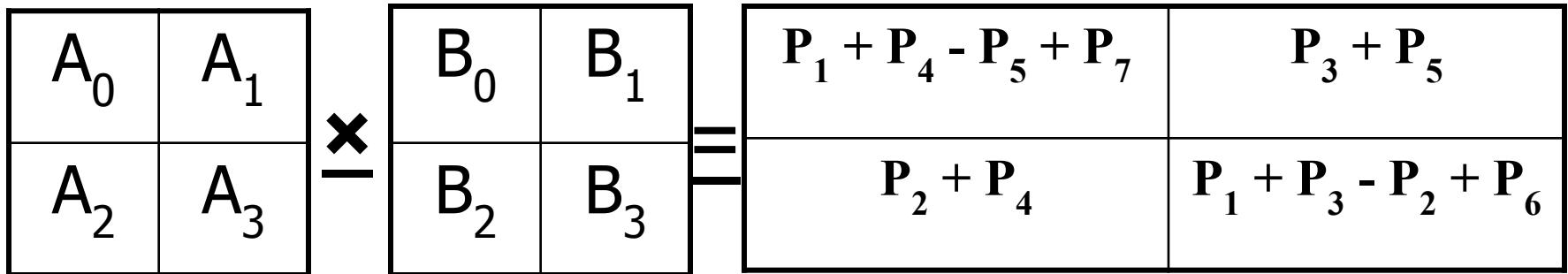
$$C_{12} = P_3 + P_5$$

$$C_{21} = P_2 + P_4$$

$$C_{22} = P_1 + P_3 - P_2 + P_6$$

Divide and Conquer Matrix Multiply

$$A \times B = R$$



- Divide matrices into sub-matrices: A₀, A₁, A₂ etc
- Use blocked matrix multiply equations
- Recursively multiply sub-matrices

$$T(n) = \begin{cases} b & n \leq 2 \\ 7T(n/2) + cn^2 & n > 2 \end{cases}$$

$$\text{Hence } T(n) = O(n^{2.81})$$

Comparison

$$C_{11} = P_1 + P_4 - P_5 + P_7$$

$$= (A_{11} + A_{22})(B_{11} + B_{22}) + A_{22} * (B_{21} - B_{11}) - (A_{11} + A_{12}) * B_{22} + (A_{12} - A_{22}) * (B_{21} + B_{22})$$

$$= A_{11}B_{11} + A_{11}B_{22} + A_{22}B_{11} + A_{22}B_{22} + A_{22}B_{21} - A_{22}B_{11} - A_{11}B_{22} - A_{12}B_{22} + A_{12}B_{21} + A_{12}B_{22} - A_{22}B_{21} - A_{22}B_{22}$$

$$= A_{11}B_{11} + A_{12}B_{21}$$



Indian Institute of Information Technology
Design and Manufacturing, Kancheepuram

Chennai 600 127, India

An Autonomous Institute under MHRD, Govt of India

An Institute of National Importance

COM 501 Advanced Data Structures and Algorithms - Lecture Notes

Recurrence Relations

In previous lectures we have discussed asymptotic analysis of algorithms and various properties associated with asymptotic notation. As many algorithms are recursive in nature, it is natural to analyze algorithms based on recurrence relations. Recurrence relation is a mathematical model that captures the underlying time-complexity of an algorithm. In this lecture, we shall look at three methods, namely, substitution method, recurrence tree method, and Master theorem to analyze recurrence relations. Solutions to recurrence relations yield the time-complexity of underlying algorithms.

1 Substitution method

Consider a computational problem P and an algorithm that solves P . Let $T(n)$ be the worst-case time complexity of the algorithm with n being the input size. Let us discuss few examples to appreciate how this method works. For searching and sorting, $T(n)$ denotes the number of comparisons incurred by an algorithm on an input size n .

Case studies: searching and sorting

Linear Search

Input: Array A , an element x

Question: Is $x \in A$

The idea behind linear search is to search the given element x linearly (sequentially) in the given array. A recursive approach to linear search first searches the given element in the first location, and if not found it recursively calls the linear search with the modified array without the first element i.e., the problem size reduces by one in the subsequent calls. Let $T(n)$ be the number of comparisons (time) required for linear search on an array of size n . Note when $n = 1$, $T(1) = 1$. Then, $T(n) = 1 + T(n - 1) = 1 + \dots + 1 + T(1)$ and $T(1) = 1$ Therefore, $T(n) = n - 1 + 1 = n$, i.e., $T(n) = \Theta(n)$.

Binary search

Input: Sorted array A of size n , an element x to be searched

Question: Is $x \in A$

Approach: Check whether $A[n/2] = x$. If $x > A[n/2]$, then prune the lower half of the array, $A[1, \dots, n/2]$. Otherwise, prune the upper half of the array. Therefore, pruning happens at every iterations. After each iteration the problem size (array size under consideration) reduces by half. Recurrence relation is $T(n) = T(n/2) + O(1)$, where $T(n)$ is the time required for binary search in an array of size n . $T(n) = T\left(\frac{n}{2^k}\right) + 1 + \dots + 1$

Since $T(1) = 1$, when $n = 2^k$, $T(n) = T(1) + k = 1 + \log_2(n)$.

$\log_2(n) \leq 1 + \log_2(n) \leq 2 \log_2(n) \forall n \geq 2$.

$T(n) = \Theta(\log_2(n))$.

Similar to binary search, the ternary search compares x with $A[n/3]$ and $A[2n/3]$ and the problem size reduces to $n/3$ for the next iteration. Therefore, the recurrence relation is $T(n) = T(n/3) + 2$, and $T(2) = 2$. Note that there are two comparisons done at each iteration and due to which additive factor '2' appears in $T(n)$.

$$\begin{aligned} T(n) &= T(n/3) + 2; \Rightarrow T(n/3) = T(n/9) + 2 \\ \Rightarrow T(n) &= T(n/(3^k)) + 2 + 2 + \dots + 2 \quad (2 \text{ appears } k \text{ times}) \end{aligned}$$

When $n = 3^k$, $T(n) = T(1) + 2 \times \log_3(n) = \Theta(\log_3(n))$

Further, we highlight that for k -way search, $T(n) = T\left(\frac{n}{k}\right) + k - 1$ where $T(k - 1) = k - 1$. Solving this, $T(n) = \Theta(\log_k(n))$.

It is important to highlight that in asymptotic sense, binary search, ternary search and k -way search (fixed k) are having same complexity as $\log_2 n = \log_2 3 \cdot \log_3 n$ and $\log_2 n = \log_2 k \cdot \log_k n$. So, $\theta(\log_2 n) = \theta(\log_3 n) = \theta(\log_k n)$, for fixed k .

In the above analysis of binary and ternary search, we assumed that $n = 2^k$ ($n = 3^k$). Is this a valid assumption? Will the above analysis hold good if $n \neq 2^k$. There is a simple fix to handle input samples which are not 2^k for any k . If the input array of size n is such that $n \neq 2^k$ for any k , then we augment least number of dummy values to make $n = 2^k$. By doing so, we are only increasing the size by at most $2n$. For ternary search, the array size increases by at most $3n$. This approximation does not change our asymptotic analysis as the search time would be one more than the actual search time.

Sorting

To sort an array of n elements using find-max (returns maximum) as a black box.

Approach: Repeatedly find the maximum element and remove it from the array. The order in which the maximum elements are extracted is the sorted sequence. The recurrence for the above algorithm is,

$$\begin{aligned} T(n) &= T(n - 1) + n - 1 = T(n - 2) + n - 2 + n - 1 = T(1) + 1 + \dots + n - 1 = \frac{(n - 1)n}{2} \\ T(n) &= \Theta(n^2) \end{aligned}$$

Merge Sort

Approach: Divide the array into two equal sub arrays and sort each sub array recursively. Do the sub-division operation recursively till the array size becomes one. Trivially, the problem size one is sorted and when the recursion bottoms out two sub problems of size one are combined to get a sorting sequence of size two, further, two sub problems of size two (each one is sorted) are combined to get a sorting sequence of size four, and so on. We shall see the detailed description of merge sort when we discuss divide and conquer paradigm. The recurrence for the merge sort is,

$$T(n) = 2T\left(\frac{n}{2}\right) + n - 1 = 2\left[2T\left(\frac{n}{2^2}\right) + \frac{n}{2} - 1\right] + n - 1$$

$$\Rightarrow 2^k T\left(\frac{n}{2^k}\right) + n - 2^k + n - 2^{k-1} + \dots + n - 1.$$

$$\text{When } n = 2^k, T(n) = 2^k T(1) + n + \dots + n - [2^{k-1} + \dots + 2^0]$$

$$\text{Note that } 2^{k-1} + \dots + 2^0 = \frac{2^{k-1+1} - 1}{2 - 1} = 2^k - 1 = n - 1$$

Also, $T(1) = 0$ as there is no comparison required if $n = 1$. Therefore, $T(n) = n \log_2(n) - n + 1 = \Theta(n \log_2(n))$

Heap sort

This sorting is based on the data structure *max-heap* which we shall discuss in detail at a later chapter. Creation of max-heap can be done in linear time. The approach is to delete the maximum element repeatedly and set right the heap to satisfy max-heap property. This property maintenance incur $O(\log n)$ time. The order in which the elements are deleted gives the sorted sequence. Number of comparisons needed for deleting an element is equal to at most the height of the max-heap, which is $\log_2(n)$. Therefore the recurrence for heap sort is,

$$T(n) = T(n - 1) + \log_2(n) \text{ where } T(1) = 0 \Rightarrow T(n) = (T(n - 2) + \log_2(n - 1)) + \log_2(n)$$

By substituting further, $\Rightarrow T(n) = T(1) + \log 2 + \log 3 + \log 4 + \dots + \log n$

$$\Rightarrow T(n) = \log(2 \cdot 3 \cdot 4 \dots n) \Rightarrow T(n) = \log(n!)$$

$\Rightarrow \log(n!) \leq n \log n$ as $n! \leq n^n$ (Stirling's Approximation)

$$\Rightarrow T(n) = \Theta(n \log_2(n))$$

Recurrence relation using change of variable technique

Problem: 1 $T(n) = 2 * T(\sqrt{n}) + 1$ and $T(1) = 1$

Introduce a change of variable by letting $n = 2^m$.

$$\Rightarrow T(2^m) = 2 * T(\sqrt{2^m}) + 1$$

$$\Rightarrow T(2^m) = 2 * T(2^{m/2}) + 1$$

Let us introduce another change by letting $S(m) = T(2^m)$

$$\Rightarrow S(m) = 2 * S(m/2) + 1$$

$$\Rightarrow S(m) = 2 * (2 * S(m/4) + 1) + 1$$

$$\Rightarrow S(m) = 2^2 * S(m/2^2) + 2 + 1$$

By substituting further,

$$\Rightarrow S(m) = 2^k * S(m/2^k) + 2^{k-1} + 2^{k-2} + \dots + 2 + 1$$

To simplify the expression, assume $m = 2^k$

$\Rightarrow S(m/2^k) = S(1) = T(2)$. Since $T(n)$ denote the number of comparisons, it has to be an integer always. Therefore, $T(2) = 2 * T(\sqrt{2}) + 1$, which is approximately 3.

$$\Rightarrow S(m) = 3 + 2^k - 1 \Rightarrow S(m) = m + 2.$$

We now have, $S(m) = T(2^m) = m + 2$. Thus, we get $T(n) = m + 2$, Since $m = \log n$, $T(n) = \log n + 2$
Therefore, $T(n) = \theta(\log n)$

Problem: 2 $T(n) = 2 * T(\sqrt{n}) + n$ and $T(1) = 1$

$$\text{Let } n = 2^m \Rightarrow T(2^m) = 2 * T(\sqrt{2^m}) + 2^m$$

$$\Rightarrow T(2^m) = 2 * T(2^{m/2}) + 2^m$$

$$\text{let } S(m) = T(2^m) \Rightarrow S(m) = 2 \times S(m/2) + 2^m$$

$$\Rightarrow S(m) = 2 \times (2 \times S(m/4) + 2^{m/2}) + 2^m$$

$$\Rightarrow S(m) = 2^2 \times S(m/2^2) + 2 \cdot 2^{m/2} + 2^m$$

By substituting further, we see that

$$\Rightarrow S(m) = 2^k \times S(m/2^k).2^{m/2^k} + 2^{k-1}.2^{m/2^{k-1}} + 2^{k-2}.2^{m/2^{k-2}} + \dots + 2 \cdot 2^{m/2} + 1 \cdot 2^m$$

An easy upper bound for the above expression is;

$$S(m) \leq 2^k \times S(m/2^k).2^m + 2^{k-1}.2^m + 2^{k-2}.2^m + \dots + 2 \cdot 2^m + 1 \cdot 2^m$$

$$S(m) = 2^k \times S(m/2^k).2^m + 2^m[2^{k-1} + 2^{k-2} + \dots + 2 + 1]$$

To simplify the expression further, we assume $m = 2^k$

$$S(m) \leq 2^k S(1).2^m + 2^m(2^k - 1)$$

Since $S(1) = T(4)$, which is approximately, $T(4) = 4$.

$$S(m) \leq 4m2^m + 2^m(m - 1)$$

$$S(m) = O(m \cdot 2^m), T(2^m) = O(m \cdot 2^m), T(n) = (n \cdot \log n).$$

Is it true that $T(n) = \Omega(n \cdot \log n)$?

From the first term of the above expression, it is clear that $S(m) \geq m \cdot 2^m$, therefore, $T(n) = \Omega(n \cdot \log n)$

Problem: 3 $T(n) = 2 * T(\sqrt{n}) + \log n$ and $T(1) = 1$

$$\text{let } n = 2^m$$

$$\Rightarrow T(2^m) = 2 * T(\sqrt{2^m}) + \log(2^m)$$

$$\Rightarrow T(2^m) = 2 * T(2^{m/2}) + m$$

$$\text{let } S(m) = T(2^m)$$

$$\Rightarrow S(m) = 2 * S(m/2) + m$$

$$\Rightarrow S(m) = 2 * (2 * S(m/4) + m/2) + m$$

$$\Rightarrow S(m) = 2^2 * S(m/2^2) + m + m$$

By substituting further,

$$\Rightarrow S(m) = 2^k * S(m/2^k) + m + m + \dots + m + m$$

$$\text{let } m = 2^k \Rightarrow S(m/2^k) = S(1) = T(2) = 2$$

$$\Rightarrow S(m) = 2 + m(k - 1) + m$$

$$\Rightarrow S(m) = 2 + m \cdot k$$

$$\Rightarrow S(m) = 2 + m \log m$$

$$\Rightarrow S(m) = O(m \log m)$$

$$\Rightarrow S(m) = T(2^m) = T(n) = O(\log n \log \log n)$$

2 Recursion Tree Method

While substitution method works well for many recurrence relations, it is not a suitable technique for recurrence relations that model divide and conquer paradigm based algorithms. Recursion Tree Method is a popular technique for solving such recurrence relations, in particular for solving unbalanced recurrence relations. For example, in case of modified merge Sort, to solve a problem of size n (to sort an array of size n), the problem is divided into two problems of size $n/3$ and $2n/3$ each. This is done recursively until the problem size becomes 1.

Consider the following recurrence relation.

1. $T(n) = 2T(n/2) + 1$

Here the number of leaves $= 2^{\log n} = n$ and the sum of effort in each level except leaves is

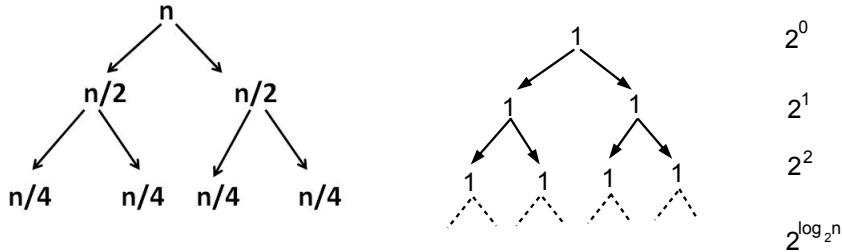


Fig. 1. The Input size reduction tree

The Corresponding Computation tree

$$\sum_{i=0}^{\log_2(n)-1} i = 2^{\log_2(n)} - 1 = n - 1$$

Therefore, the total time $= n + n - 1 = 2n - 1 = \Theta(n)$.

2. $T(n) = 2T(n/2) + 1$

$$T(1) = \log n$$

Solution: Note that $T(1)$ is $\log n$.

$$\text{Given } T(n) = 2T(n/2) + 1$$

$$= 2[2T(n/4) + 1] + 1 = 2^2T(n/2^2) + 2 + 1$$

$$= 2^2[2T(n/2^3) + 1] + 2 + 1 = 2^3T(n/2^3) + 2^2 + 2 + 1 = 2^kT(n/2^k) + (2^{k-1} + 2^{k-2} + \dots + 2 + 1)$$

We stop the recursion when $n/2^k = 1$.

$$\text{Therefore, } T(n) = 2^kT(1) + 2^k - 1 = 2^k \log n + 2^k - 1$$

$$= 2^{\log_2 n} \log n + 2^{\log_2 n} - 1$$

$$= n \log n + n - 1$$

Clearly, $n \log n$ is the significant term. $T(n) = \theta(n \log n)$.

$$3. T(n) = 3T(n/4) + cn^2$$

Note that the number of levels = $\log_4 n + 1$

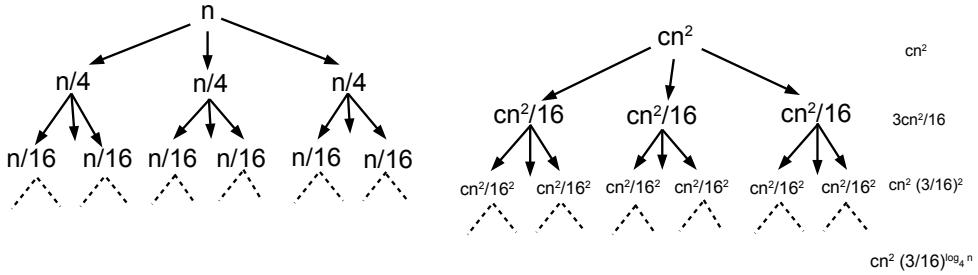


Fig. 2. Input size reduction tree

The Corresponding Computation tree

$$\text{Also the number of leaves} = 3^{\log_4 n} = n^{\log_4 3}$$

The total cost taken is the sum of the cost spent at all leaves and the cost spent at each subdivision operation. Therefore, the total time taken is $T(n) = cn^2 + \frac{3}{16}cn^2 + (\frac{3}{16})^2cn^2 + \dots + (\frac{3}{16})^{\log_4(n)-1} + \text{number of leaves} \times T(1)$.

$$\begin{aligned} T(n) &= \sum_{i=0}^{\log_4(n)-1} cn^2(\frac{3}{16})^i + n^{\log_4(3)} \times T(1) \\ &= \frac{\frac{3}{16}^{\log_4(n)} - 1}{\frac{3}{16} - 1} cn^2 + n^{\log_4(3)} \times T(1) \\ &= \frac{1 - \frac{3}{16}^{\log_4(n)}}{1 - \frac{3}{16}} cn^2 + n^{\log_4(3)} \times T(1) \\ &= \frac{1 - n^{\log_4(\frac{3}{16})}}{1 - \frac{3}{16}} cn^2 + n^{\log_4(3)} \times T(1) \\ &= d'cn^2 + n^{\log_4(3)}T(1) \text{ where the constant } d' = \frac{1 - n^{\log_4(\frac{3}{16})}}{1 - \frac{3}{16}} \end{aligned}$$

$$= d'cn^2 + n^{\log_4(3)}T(1). \text{ Therefore, } T(n) = O(n^2)$$

Since the root of the computation tree contains cn^2 , $T(n) = \Omega(n^2)$. Therefore, $T(n) = \theta(n^2)$

$$4. T(n) = T(n/3) + T(2n/3) + O(n)$$

Note that the leaves are between the levels $\log_3 n$ and $\log_{3/2} n$

From the computation tree, the cost incurred in all levels except leaves is at most $\log_{3/2} n * cn = O(n \log n)$

The Number of leaves = $2^{\log_{3/2} n} = n^{\log_{3/2} 2}$

$$\text{The Total Time } T(n) = n^{\log_{3/2} 2} + cn(\log_{3/2} n - 1) \geq n \log n + n \log n - cn = \Omega(n \log n)$$

$$\therefore T(n) = \theta(n \log n)$$

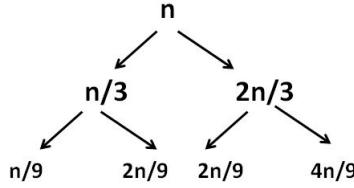


Fig. 3. Recursion Tree

$$5. T(n) = T\left(\frac{n}{10}\right) + T\left(\frac{9n}{10}\right) + n$$

Note that the leaves of the computation tree are found between levels $\log_{10}(n)$ and $\log_{\frac{10}{9}}(n)$

Assume all the leaves are at level $\log_{10}(n)$

Then $T(n) \geq n \log_{10}(n) \implies T(n) = \Omega(n \log(n))$

Assume all the leaves are at level $\log_{\frac{10}{9}}(n)$

Then $T(n) \leq n \log_{\frac{10}{9}}(n) \implies T(n) = O(n \log(n))$

Solution using Guess method

One can guess the solution to recurrence relation $T(n)$ and verify the guess by simple substitution.

For $T(n) = T(n/3) + T(2n/3) + O(n)$, Guess $T(n) \leq dn \log(n)$.

Substituting the guess, we get

$$T(n) = dn/3 \log n/3 + dn2/3 \log 2n/3 + cn$$

$$T(n) = dn/3 \log n - dn/3 \log 3 + d2n/3 \log 2n - d2n/3 \log 3 + cn$$

$$T(n) = dn \log n - dn \log 3 + d2n/3 + cn$$

Since $T(n)$ is at most $dn \log n$, we get

$$dn \log n - dn \log 3 + d2n/3 + cn \leq dn \log n \implies cn \leq dn(\log 3 - 2/3)$$

$c \leq d(\log 3 - 2/3)$ Choose 'c' and 'd' such that the inequality is respected

$$\therefore T(n) \leq dn \log n = O(n \log n)$$

$$6. T(n) = 2T(n/2) + n, T(1) = 1$$

Solution: Guess $T(n) = O(n^2)$. $T(n) \leq 2cn^2/4 + n = cn^2/2 + n \leq cn^2$ (Possible)

Therefore, $T(n) = O(n^2)$.

Guess : $T(n) = O(n \log n)$

$$T(n) \leq 2cn/2 \log(n/2) + n = cn \log n - cn + n$$

$$cn \log n - cn + n \leq cn \log n \text{ (Possible)}$$

Therefore, $T(n) = O(n \log n)$

Guess : $T(n) = O(n)$

$$T(n) \leq 2cn/2 + n$$

$$= cn + n$$

$$= cn + n \leq cn \text{ (not possible)}$$

Therefore, $T(n) \neq O(n)$

$$7. T(n) = 2T(n/2) + 1$$

Guess : $T(n) = O(\log n)$

$$\begin{aligned}
T(n) &\leq 2c \log n / 2 + 1 \\
&= 2c \log n - 2c + 1 \\
2c \log n - 2c + 1 &\leq c \log n \text{ (Not possible)} \\
\text{Therefore, } T(n) &\neq O(\log n)
\end{aligned}$$

$$\begin{aligned}
\text{Guess : } T(n) &= O(n) \\
T(n) &\leq 2cn / 2 + 1 = cn + 1 \\
cn + 1 &\leq cn \text{ (Not possible)} \\
\text{The guess that } T(n) &= O(n) \text{ may not be an appropriate guess. A careful fine tuning on the} \\
\text{above guess shows that } T(n) &\text{ is indeed } O(n).
\end{aligned}$$

3 Master Theorem : A Ready Reckoner

We shall now look at a method 'master theorem' which is a 'cook book' for many well-known recurrence relations. It presents a framework and formulae using which solutions to many recurrence relations can be obtained very easily. Almost all recurrences of type $T(n) = aT(n/b) + f(n)$ can be solved easily by doing a simple check and identifying one of the three cases mentioned in the following theorem. By comparing $n^{\log_b a}$ (the number of leaves) with $f(n)$, one can decide upon the time complexity of the algorithm.

Master Theorem Let $a \geq 1$ and $b \geq 1$ be constants, let $f(n)$ be a non negative function, and let $T(n)$ be defined on the non-negative integers by the recurrence

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

where we interpret n/b to be either $\lfloor n/b \rfloor$ or $\lceil n/b \rceil$. Then $T(n)$ has the following asymptotic bounds:

Case 1: If $f(n) = O(n^{\log_b a - \epsilon})$ for some constant $\epsilon \geq 0$ Then $T(n) = \theta(n^{\log_b a})$

Case 2: If $f(n) = \theta(n^{\log_b a})$ then $T(n) = \theta(n^{\log_b a} \log n)$

Case 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 $T(n) = \theta(f(n))$.

In the next section, we shall solve recurrence relations by applying master theorem. Later, we discuss a proof of master's theorem and some technicalities to be understood before applying master theorem.

3.1 Deducing Time Complexity using Master's Theorem: Some Examples

1. $T(n) = 9T(n/3) + n$

$$\begin{aligned}
n^{\log_b a} &= n^{\log_3 9} = n^2 \\
f(n) &= n \\
\text{Comparing } n^{\log_b a} \text{ and } f(n) \\
n &= O(n^2)
\end{aligned}$$

Satisfies Case 1 of Master's Theorem

That implies $T(n) = \Theta(n^{\log_b a}) = \Theta(n^2)$

2. $T(n) = T(2n/3) + 1$

$$n^{\log_b a} = n^{\log_3/2 1} = n^0 = 1 = f(n)$$

Comparing $n^{\log_b a}$ and $f(n)$

$$n^{\log_b a} = \Theta(f(n))$$

Satisfies Case 2 of Master's Theorem

That implies $T(n) = \Theta(n^{\log_b a} \cdot \log n) = \Theta(\log n)$

3. $T(n) = 2T(n/2) + n$

$$n^{\log_b a} = n^{\log_2 2} = n^1 = n = f(n)$$

Comparing $n^{\log_b a}$ and $f(n)$

$$n^{\log_b a} = \Theta(f(n))$$

Satisfies Case 2 of Master's Theorem

That implies $T(n) = \Theta(n^{\log_b a} \cdot \log n) = \Theta(n \log n)$

4. $T(n) = 3T(n/4) + n \log n$

$$n^{\log_b a} = n^{\log_4 3}$$

$$f(n) = n \log n$$

Comparing $n^{\log_b a}$ and $f(n)$

Satisfies Case 3 of Master's Theorem

Checking the regularity condition

$$a.f(n/b) < c.f(n) \text{ (for some constant } c < 1\text{)}$$

$$(3n/4) \log n/4 < c.n \log n$$

$$(3/4)n[\log n - \log 4] < (3/4)n \log n \text{ where (c=3/4)}$$

That implies the regularity condition is satisfied

That implies $T(n) = \Theta(f(n)) = \Theta(n \log n)$

5. $T(n) = 4T(n/2) + n$

$$n^{\log_b a} = n^{\log_2 4} = n^2$$

$$f(n) = n$$

Comparing $n^{\log_b a}$ and $f(n)$

$$n = O(n^2)$$

Satisfies Case 1 of Master's Theorem

That implies $T(n) = \Theta(n^{\log_b a}) = \Theta(n^2)$

6. $T(n) = 4T(n/2) + n^2$

$$n^{\log_b a} = n^{\log_2 4} = n^2 = f(n)$$

Comparing $n^{\log_b a}$ and $f(n)$

$$n^{\log_b a} = \Theta(f(n))$$

Satisfies Case 2 of Master's Theorem

That implies $T(n) = \Theta(n^{\log_b a} \cdot \log n) = \Theta(n^2 \log n)$

7. $T(n) = 2T(n/2) + n \log n$

$$n^{\log_b a} = n^{\log_2 2} = n^1 = n$$

Comparing $n^{\log_b a} = n$ and $f(n) = n \log n$

Doesn't Satisfy either case 1 or 2 or 3 of the master's theorem

Case 3 states that $f(n)$ should be polynomially larger but here it is asymptotically larger than $n^{\log_b a}$ only by a factor of $\log n$

Note: If $f(n)$ is polynomially larger than $g(n)$, then $\frac{f(n)}{g(n)} = n^\epsilon, \epsilon > 0$. Note that in the above recurrence $n \log n$ is asymptotically larger than $n^{\log_b a}$ but not polynomially larger. i.e., $n = O(n \log n)$ whereas $\frac{n}{n \log n} \neq n^\epsilon$, for any $\epsilon > 0$

$$8. T(n) = 4T(n/2) + n^2 \log n$$

$$n^{\log_b a} = n^{\log_2 4} = n^2$$

$$\text{Comparing } n^{\log_b a} = n^2 \text{ and } f(n) = n^2 \log n$$

Doesn't Satisfy either case 1 or 2 or 3 of the master's theorem

Case 3 states that $f(n)$ should be polynomially larger but here it is asymptotically larger than $n^{\log_b a}$ by a factor of $\log n$

If recurrence relations fall into the gap between case 1 and case 2 or case 2 and case 3, master theorem can not be applied and such recurrences can be solved using recurrence tree method.

Technicalities:

- From the above examples, it is clear that in each of the three cases, we compare the function $f(n)$ with the function $n^{\log_b a}$. Intuitively, the larger of the two functions determines the solution to the recurrence. If, as in case 1, the function $n^{\log_b a}$ is the larger, then the solution is $T(n) = \theta(n^{\log_b a})$.
- If, as in case 3, the function $f(n)$ is the larger, then the solution is $T(n) = \theta(f(n))$.
- If, as in case 2, the two functions are the same size, we multiply by a logarithmic factor which comes from the height of the tree, and the solution is $T(n) = \theta(n^{\log_b a} \log n)$.
- Also, in the first case, not only must $f(n)$ be smaller than $n^{\log_b a}$, it must be *polynomially* smaller. That is, $f(n)$ must be asymptotically smaller than $n^{\log_b a}$ by a factor of n^ϵ for some constant $\epsilon > 0$.
- In the third case, not only must $f(n)$ be larger than $n^{\log_b a}$, it also must be polynomially larger and in addition satisfy the regularity condition that $af(n/b) \leq cf(n)$.
- Note that the three cases do not cover all the possibilities for $f(n)$. There is a gap between cases 1 and 2 when $f(n)$ is smaller than $n^{\log_b a}$ but not polynomially smaller.
- Similarly, there is a gap between cases 2 and 3 when $f(n)$ is larger than $n^{\log_b a}$ but not polynomially larger. If the function $f(n)$ falls into one of these gaps, or if the regularity condition in case 3 fails to hold, you cannot use the master method to solve the recurrence.
- Here is a recurrence relation which satisfies the case 3 of master theorem but fails to satisfy regularity condition.

$$1. T(n) = 2T(\frac{n}{2}) + n^2(1 + \sin(n))$$

Here $f(n) = n^2(1 + \sin(n))$, $n^2(1 + \sin(n)) \leq c.n^{\log_2 2+\epsilon}$ where $\epsilon = 1$. Therefore, case 3 of master's theorem applies. Now we shall check the regularity lemma.

$2.(\frac{n}{2})^2(1 + \sin(\frac{n}{2})) \leq c.n^2(1 + \sin(n))$. Note that for every odd value m , $\sin(\frac{2m+1}{2}\pi) = -1$

and $\sin\left(\frac{2m+1}{4}\pi\right) = -\frac{1}{\sqrt{2}}$. When $n = \frac{2m+1}{2}$, it implies that $\frac{1}{2} \cdot n^2 \left(1 - \frac{1}{\sqrt{2}}\right) \leq c \cdot n^2 (1 - 1)$ and such a c does not exist.

- $$2. \ T(n) = 2T\left(\frac{n}{2}\right) + n^2 \cdot 2^{-n}$$

Here $f(n) = n^2 \cdot 2^{-n} = O(n^{\log_2 2 + \epsilon})$, $\epsilon = 1$. For checking regularity lemma,

$2 \cdot (\frac{n}{2})^2 \cdot 2^{-\frac{n}{2}} \leq c \cdot n^2 \cdot 2^{-n}$. It follows that $\frac{1}{2} 2^{-\frac{n}{2}} \leq c \cdot 2^{-n} \implies c \geq \frac{1}{2} \cdot 2^{\frac{n}{2}}$, which is not possible as $c < 1$.

3.2 Proof of Master’s Theorem

Lemma 1 Let $a \geq 1$ & $b > 1$ be constants & let $f(n)$ be a non-negative function defined on exact powers of b . A function $g(n)$ defined over exact powers of b by

$$g(n) = \sum_{j=0}^{\log_b n - 1} a^j f(n/b^j)$$

can be then bounded asymptotically for exact powers of b , as follows.

1. If $f(n) = O(n^{\log_b a - \epsilon})$ for some constant $\epsilon \geq 0$, then $g(n) = O(n^{\log_b a})$
 2. If $f(n) = \theta(n^{\log_b a})$, then $g(n) = \theta(n^{\log_b a} f(n))$
 3. If $af(n/b) \leq cf(n)$ for some constant $c \leq 1$, & $\forall n \geq b$, then $g(n) = \theta(f(n))$

Lemma 2 Let $a \geq 1$ & $b \geq 1$ let $f(n)$ be a non-negative function defined on exact powers of 'b'. Define $T(n)$ on exact powers of b by the recurrence by.

$$T(n) = \begin{cases} \theta(1) & \text{if } n = 1 \\ aT(n/b) + f(n) & \text{if } n = b^i \end{cases}$$

Then, $T(n) = \theta(n^{\log_b a}) + \sum_{j=0}^{\log_b n-1} a^j f(n/b^j)$

At level i , there are a^i subproblems each of complexity $f(n/b^i)$

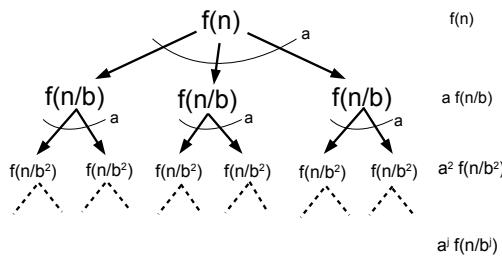


Fig. 4. Recursion Tree

$$\text{Number of leaves} = a^{\log_b n} = n^{\log_b a}$$

$$\text{Cost} = \sum_{j=0}^{\log_b n - 1} a^j f(n/b^j)$$

The first term is from the leaves of the tree & the second term is summation of every level other than the last.

$$\Rightarrow T(n) = \theta(n^{\log_b a}) + \sum_{j=0}^{\log_b n - 1} a^j f(n/b^j)$$

We now present a proof. Here we analyze the recurrence relation, under the simplifying assumption that $T(n)$ is defined only on exact powers of $b > 1$, that is, for $n = 1, b, b^2 \dots$. From *Lemma 2*

$$T(n) = \theta(n^{\log_b a}) + \sum_{j=0}^{\log_b n-1} a^j f(n/b^j)$$

Case 1: $f(n)=O(n^{\log_b a-\epsilon})$

Recall that $g(n)=\sum_{j=0}^{\log_b n-1} a^j f(n/b^j)$

Since $f(n/b^j)=O((n/b^j)^{\log_b a-\epsilon})$, $g(n)=O(\sum_{j=0}^{\log_b n-1} a^j (n/b^j)^{\log_b a-\epsilon})$

On rearranging, $g(n)=\sum_{j=0}^{\log_b n-1} a^j n^{\log_b a-\epsilon} / (b^{\log_b a-\epsilon})^j \implies g(n)=n^{\log_b a-\epsilon} \sum_{j=0}^{\log_b n-1} a^j (b^\epsilon)^j / ((b^{\log_b a})^j)$

$$\implies g(n)=n^{\log_b a-\epsilon} \sum_{j=0}^{\log_b n-1} a^j (b^\epsilon)^j / a^j \implies g(n)=n^{\log_b a-\epsilon} \sum_{j=0}^{\log_b n-1} (b^\epsilon)^j$$

$$\implies g(n)=n^{\log_b a-\epsilon} [(b^\epsilon)^{\log_b n} - 1/(b^\epsilon - 1)] \implies g(n)=n^{\log_b a-\epsilon} [(n^\epsilon - 1)/(b^\epsilon - 1)]$$

\implies Since $b > 1$, and $\epsilon > 1 \implies b^\epsilon - 1$ is constant (say c)

$$g(n) \leq c \cdot n^{\log_b a-\epsilon} \cdot n^\epsilon \implies g(n)=O(n^{\log_b a})$$

Finally, $T(n)=\theta(n^{\log_b a}) + O(n^{\log_b a})$ and $T(n)=\theta(n^{\log_b a})$

Case 2: $f(n)=\theta(n^{\log_b a})$

Recall, $g(n)=\sum_{j=0}^{\log_b n-1} a^j f(n/b^j)$

Since $f(n/b^j)=\theta((n/b^j)^{\log_b a})$, $g(n)=\sum_{j=0}^{\log_b n-1} a^j (n/b^j)^{\log_b a}$

$$g(n)=\sum_{j=0}^{\log_b n-1} a^j n^{\log_b a} / ((b^j)^{\log_b a}) \implies g(n)=n^{\log_b a} \sum_{j=0}^{\log_b n-1} a^j 1 / ((b^{\log_b a})^j)$$

$$\implies g(n)=n^{\log_b a} \sum_{j=0}^{\log_b n-1} a^j / a^j \implies g(n)=n^{\log_b a} \sum_{j=0}^{\log_b n-1} 1$$

$$\implies g(n)=n^{\log_b a} \theta(\log_b n) \implies g(n)=\theta(n^{\log_b a} \log_b n)$$

Therefore, $T(n)=\theta(n^{\log_b a}) + \theta(n^{\log_b a} \log_b n)$ and $T(n)=\theta(n^{\log_b a} \log_b n)$

Case 3: $f(n)=\Omega(n^{\log_b a+\epsilon})$

Recall, $g(n)=\sum_{j=0}^{\log_b n-1} a^j f(n/b^j)$

Given a $f(n/b) \leq c f(n) \implies f(n/b) \leq c/a f(n)$

After j iterations, $f(n/b^j) \leq (c/a)^j f(n)$

$$\begin{aligned} g(n) &\leq \sum_{j=0}^{\log_b n - 1} a^j (c^j/a^j) f(n) \implies g(n) \leq \sum_{j=0}^{\log_b n - 1} c^j f(n) \\ &\implies g(n) \leq \sum_{j=0}^{\infty} c^j f(n) \implies g(n) \leq f(n) \times 1/(1 - c) \\ &\implies \therefore g(n) = O(f(n)) \end{aligned}$$

Since $g(n) = f(n) + \sum_{j=1}^{\log_b n - 1} a^j f(n/b^j)$, $g(n) > f(n)$ and $g(n) = \Omega(f(n))$.
 $\therefore g(n) = \theta(f(n))$

$$T(n) = \theta(n^{\log_b a}) + \theta(f(n))$$

$$\text{Since, } f(n) = \Omega(n^{\log_b a + \epsilon}), T(n) = \theta(f(n))$$

This completes a proof of the Master's theorem.

Not all recurrence relations can be solved using recurrence tree, masters theorem and substitution method. We here mention a method from difference equation to solve homogenous recurrence relations. That is, recurrence relations which depends on r previous iterations (terms). Particularly, recurrence relations of the form $T(n) = c_1 T(n-1) + c_2 T(n-2) + \dots + c_r T(n-r)$. In the next section, we shall discuss a method for solving well-known recurrences like 'Fibonacci series' using 'characteristic equation' based approach.

3.3 Solution using Characteristic Equation Approach

$$1. a_n - 7a_{n-1} + 10a_{n-2} = 4^n$$

$$T(n) - 7T(n-1) + 10T(n-2) = 4^n$$

$$\text{Characteristic equation is } x^2 - 7x + 10 = 0$$

$$(x-5)(x-2) = 0; x = 5, 2$$

$$\text{Complementary function (C.F.) is } c_1 5^n + c_2 2^n$$

$$\text{Solution is } a_n = c_1 5^n + c_2 2^n + \text{Particular integral (P.I.)}$$

$$\text{For finding P.I, guess } a_n = d4^n$$

$$d4^n - 7d4^{n-1} + 10d4^{n-2} = 4^n$$

$$d - \frac{7}{4}d + \frac{10}{16}d = 1 \implies d = -8$$

$$\text{Therefore solution is } T(n) = c_1 5^n + c_2 2^n - 8 \cdot 4^n$$

$$2. a_n - 4a_{n-1} + 4a_{n-2} = 2^n$$

$$T(n) - 4T(n-1) + 4T(n-2) = 2^n$$

$$\text{Characteristic equation is } x^2 - 4x + 4 = 0 \text{ and the roots are } x = 2, 2$$

$$\text{Complementary function (C.F.) is } c_1 \cdot 2^n + c_2 \cdot n \cdot 2^n$$

For P.I, note that the guesses $d \cdot 2^n$ and $d \cdot n \cdot 2^n$ will not work as the roots and base of the exponent (non-homogenous) term are same. Therefore we guess $d \cdot n^2 \cdot 2^n$

$$d \cdot n^2 \cdot 2^n - 4d(n-1)^2 \cdot 2^{n-1} + d \cdot (n-2)^2 \cdot 2^{n-2} = 2^n$$

$$d \cdot n^2 - 2d(n^2 - 2n + 1) + d \cdot (n^2 - 4n + 4) = 1$$

$$\text{Simplifying we get } d = \frac{1}{2}$$

$$\text{Therefore } T(n) = c_1 2^n + c_2 \cdot n \cdot 2^n + \frac{n^2 \cdot 2^n}{2}$$

Exercise

1. Solve the following

$$T(n) = 4T\left(\frac{n}{4}\right) + n^2$$

$$T(n) = 4T\left(\frac{n}{2}\right) + n^2$$

$$T(n) = T(n+5) + n$$

$$T(n) = 3T\left(\frac{n}{4}\right) + n$$

$$T(n) = 2T\left(\frac{n}{2}\right) + 1$$

$$T(n) = 2T\left(\frac{n}{2} + 17\right) + n$$

2. Can Master method be applied to $T(n) = 8T(n/2) + n^3 \log n$. If not why?

Acknowledgements: Lecture contents presented in this module and subsequent modules are based on the following text books and most importantly, author has greatly learnt from lectures by algorithm exponents affiliated to IIT Madras/IMSc; Prof C. Pandu Rangan, Prof N.S.Narayanaswamy, Prof Venkatesh Raman, and Prof Anurag Mittal. Author sincerely acknowledges all of them. Special thanks to Teaching Assistants Mr.Renjith.P and Ms.Dhanalakshmi.S for their sincere and dedicated effort and making this scribe possible. Author has benefited a lot by teaching this course to senior undergraduate students and junior undergraduate students who have also contributed to this scribe in many ways. Author sincerely thanks all of them.

References:

1. E.Horowitz, S.Sahni, S.Rajasekaran, Fundamentals of Computer Algorithms, Galgotia Publications.
2. T.H. Cormen, C.E. Leiserson, R.L.Rivest, C.Stein, Introduction to Algorithms, PHI.
3. Sara Baase, A.V.Gelder, Computer Algorithms, Pearson.

Unit III

Dynamic Programming : General method, applications, optimal binary search trees, 0/1 knapsack problem, All pairs shortest path problem, travelling salesperson problem, optimal rod-cutting-Top down approach and bottom up approach.

Dynamic Programming

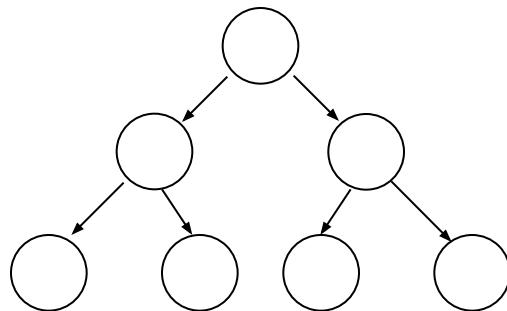
- Dynamic Programming is an algorithm design technique that can be used when the solution to a problem may be viewed as the result of a sequence of decisions.
- Dynamic Programming is an algorithm design technique for *optimization problems*: often minimizing or maximizing the objective function.
- Like divide and conquer, DP solves problems by combining solutions to sub-problems.
- Unlike divide and conquer, sub-problems are not independent.
 - Sub-problems may share sub-sub-problems

Optimization Problems

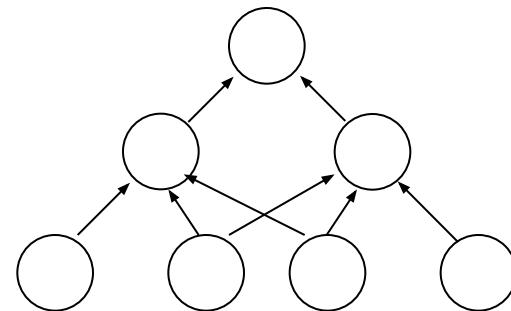
- Dynamic programming is typically to solve **optimization problems**
 - There can be **many possible solutions**.
 - Each solution **has a value**.
 - We find a solution with ***the* optimal** (minimum or maximum) **value**.
 - Such a solution is called ***an* optimal solution** to the problem.
- To Enumerate all possible values and find an optimal solution which is known as a naive approach would take exponential time.
- Dynamic Programming solves different types of problems in time Polynomial Time.

Dynamic Programming

- In dynamic programming we usually **reduce time** by increasing the **amount of space**
- We **solve** the problem by solving **sub-problems of increasing size** and **saving** each optimal solution in a **table** (usually).
- The table is then used for finding the **optimal solution** to larger problems.
- Time is saved since each sub-problem is solved only once.
- In other words
 - DP algorithm solves every sub-problem just once and **saves its answer in a table** and then reuse it.



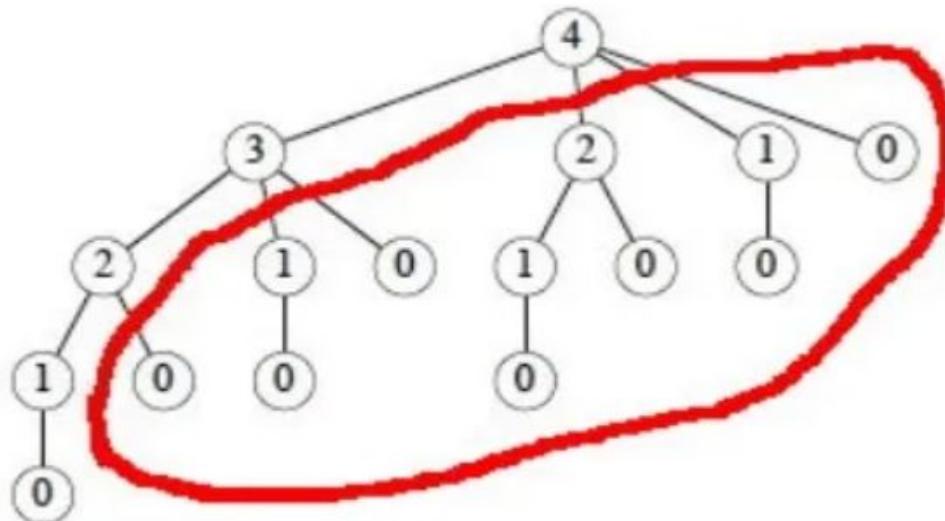
Divide-and-Conquer



Dynamic Programming

Overlapping Subproblem

- Subproblems are smaller versions of the original problem. Any problem has overlapping sub-problems if finding its solution involves solving the same subproblem multiple times.
- **For ex:** F4 is divided in to F3 ,F2,F1 and F0 and again F3 is further divided in to F2,F1 and F0. So, here F2 should be computed twice hence it is an overlapping sub problem. F1 and F0 are also overlapping subproblems in the given figure.



Optimal substructure

- Any problem has optimal substructure property if its overall optimal solution can be constructed from the optimal solutions of its subproblems.

Principle of optimality

- **Principle of optimality:** An optimal sequence of decisions has the property that whatever the initial state and initial decision are, the remaining decisions must constitute an optimal decision sequence with regard to the state resulting from the first decision.

or in other words

- *In optimal sequence of decisions, the sub sequence must also be optimal.* Suppose that in solving a problem, we have to make a sequence of decisions D_1, D_2, \dots, D_n . If this sequence is optimal, then any k sequence of decisions, $1 < k < n$ must also be optimal.
- *e.g. the shortest path problem*

If i, i_1, i_2, \dots, j is a shortest path from i to j , then $, i_{k-1}, i_k, \dots, j$ must be a shortest path from i_{k-1} to j

Applications of DP

The versatility of the dynamic programming method is really appreciated by exposure to a wide variety of applications.

- Information theory.
- Control theory.
- Bioinformatics.
- Operations research.
- Computer science :
theory, graphics, Artificial Intelligence, etc.

Dynamic Programming

- The best way to get a feel for this is through some examples.
 - Optimal Binary Search
 - 0-1 Knapsack Problem
 - Travelling Sales Person Problem
 - All Pairs Shortest
 - Optimal Rod Cutting problem

Elements of Dynamic Programming

- The development of a dynamic-programming algorithm can be broken into a sequence of four steps.
 1. Characterize the structure of an optimal solution.
 2. Recursively define the value of an optimal solution.
 3. Compute the value of an optimal solution in a bottom-up fashion.
 4. Construct an optimal solution from computed information.

The Idea of Developing a DP Algorithm

Step1: **Structure:** Characterize the structure of an optimal solution.

- Decompose the problem into smaller problems, and find a relation between the structure of the optimal solution of the original problem and the solutions of the smaller problems.

Step2: **Principle of Optimality:** Recursively define the value of an optimal solution.

- Express the solution of the original problem in terms of optimal solutions for smaller problems.

Step 3: Bottom-up computation: Compute the value of an optimal solution in a bottom-up fashion by using a table structure.

Step 4: Construction of optimal solution: Construct an optimal solution from computed information.

Steps 3 and 4 may often be combined.

Binary Search Tree

- A Binary search Tree is a binary tree. It may be empty. If it is not empty, then it satisfies the following properties.
 - Every element has a key and no two elements have the same key.
 - The keys(if any) in the left subtree are smaller than the key in the root.
 - The keys(if any) in the right subtree are larger than the key in the root.
 - The left and right subtrees are also binary search trees.

Example

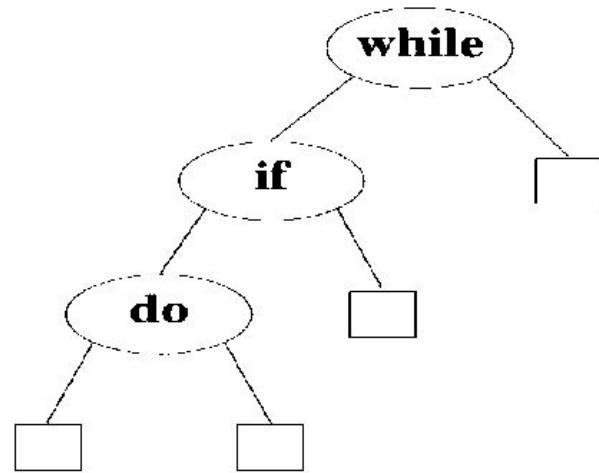
- Example : the given identifier set is $(a_1, a_2, a_3) = (\text{do}, \text{if}, \text{while})$.
- The no. of BST that can be formed using $n=3$ is

$$1/(n+1) 2^n C_n$$

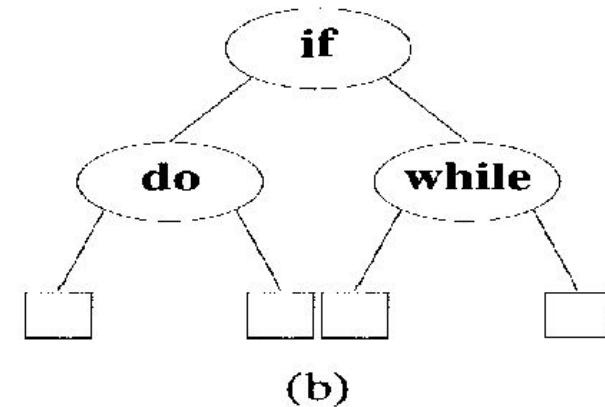
$$nC_r = n! / (r! * (n - r)!)$$

When $n=3$, no. of BST trees are $\frac{1}{4}((2^3)C_3) = 5$

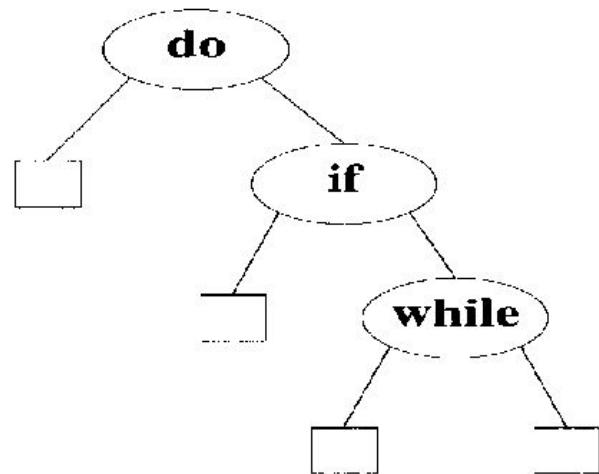
Different possible BST with n=3



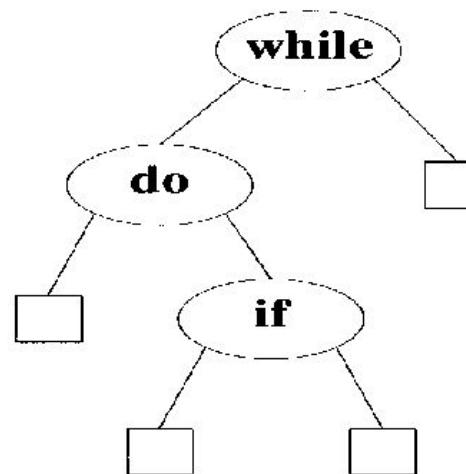
(a)



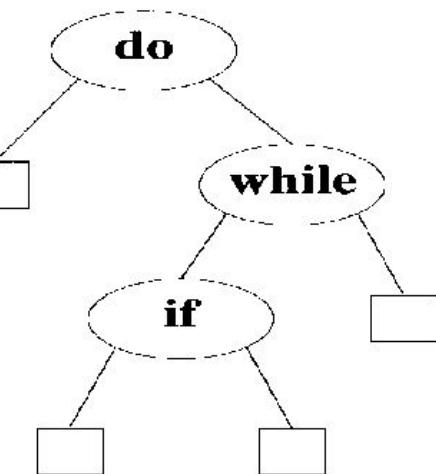
(b)



(c)



(d)



(e)

Optimal binary search trees

- Given sequence $S = a_1 < a_2 < \dots < a_n$ of n sorted keys, with a search probability p_i for each key a_i , build a binary search tree (BST) **with minimum expected search cost.**
- Let $\{a_1, a_2, \dots, a_n\}$ be a set of n identifiers : $a_1 < a_2 < \dots < a_n$
- Let P_i , $1 \leq i \leq n$: the probability with which a_i is searched. **(successful Search)**
- Let Q_i , $0 \leq i \leq n$: the probability that the search fails.
Let x be search element where $a_i < x < a_{i+1}$ ($a_0 = -\infty$, $a_{n+1} = \infty$).
(unsuccessful Search)

$$\sum_{i=1}^n P_i + \sum_{i=0}^n Q_i = 1$$

Cost Function

- In obtaining a cost function for BST, add a fictitious node in place of every empty sub tree in search tree, such nodes are called as external nodes.
- If a BST represents n identifiers, then there are $n+1$ external nodes and n internal nodes.
- Every internal node represents a point where a successful search may terminate. Every external node represents a point where an unsuccessful search may terminate.
- The cost contribution from the internal node a_i is

$$p(i) * \text{level}(a_i)$$

Cost BST

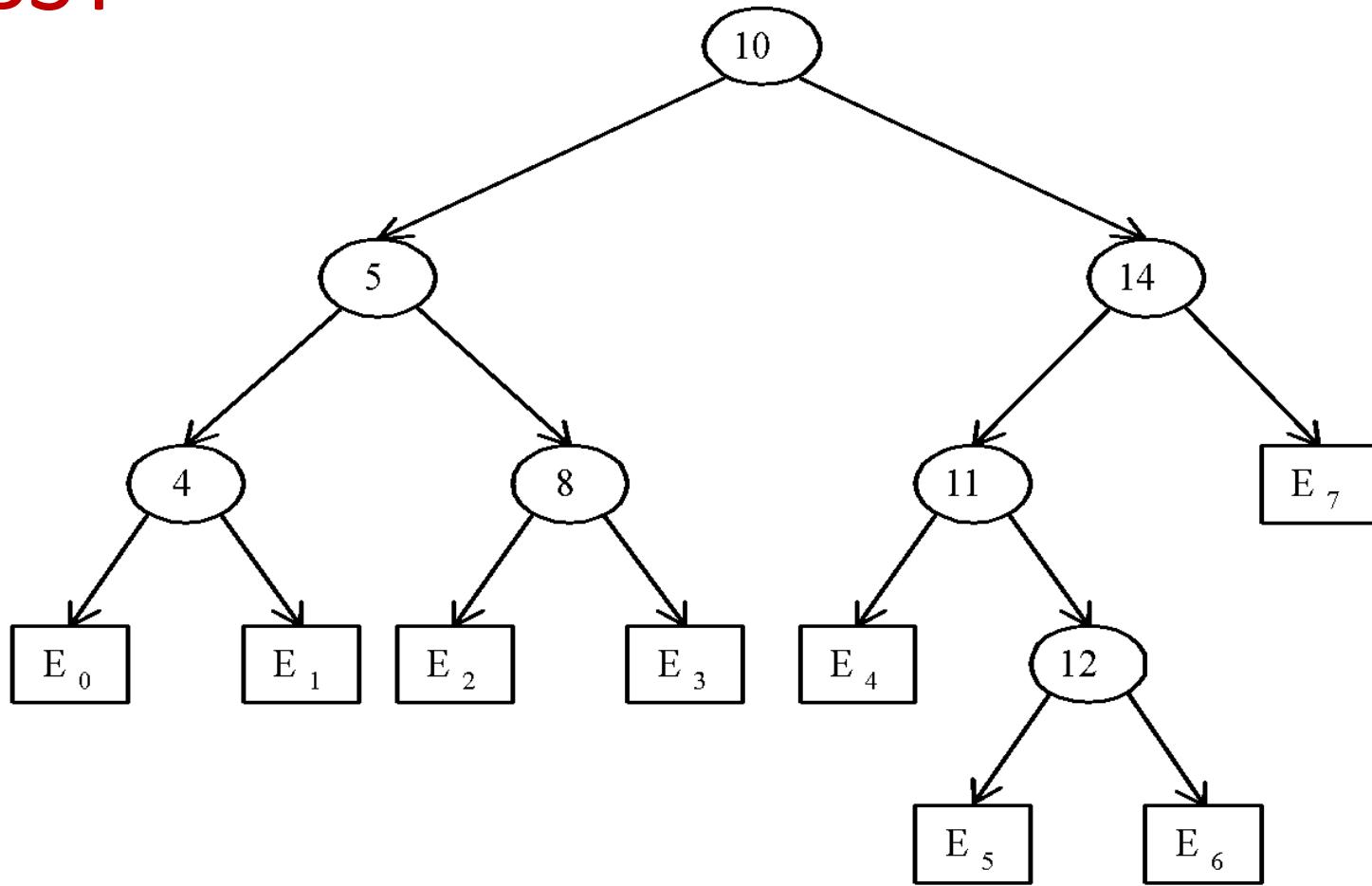
- The identifiers not in the binary search tree can be partitioned into $n + 1$ equivalence classes E_i , $0 \leq i \leq n$.
- The class E_0 contains all identifiers x such that $x < a_1$.
- The class E_i contains all identifiers x such that $a_i < x < a_{i+1}$, $1 \leq i < n$.
The class E_n contains all identifiers x , $x > a_n$.
- for all identifiers in the same class E_i , the search terminates at the same external node.
- For identifiers in different E_i the search terminates at different external nodes.
- If the failure node for E_i is at level l , the cost of this node is
$$q(i) * (\text{level}(E_i) - 1)$$

- The expected cost of a binary tree is computed using:

$$\sum_{i=1}^n P_i * \text{level}(a_i) + \sum_{i=0}^n Q_i * (\text{level}(E_i) - 1)$$

- The level of the root : 1

Cost BST



- Identifiers : 4, 5, 8, 10, 11, 12, 14
- Internal node : **successful search, P_i**
- External node : **unsuccessful search, Q_i**

Example

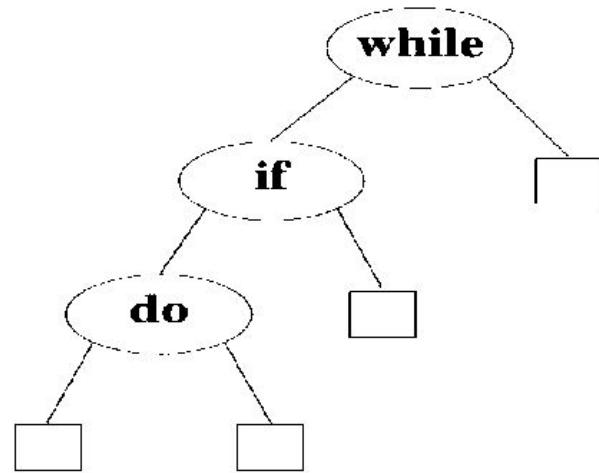
- Example : the given identifier set is $(a_1, a_2, a_3) = (\text{do}, \text{if}, \text{while})$. With equal probabilities $p(i) = q(i) = 1/7$ for all i , find the optimal binary search tree.
- The no. of BST that can be formed using $n=3$ is

$$1/(n+1) 2^n C_n$$

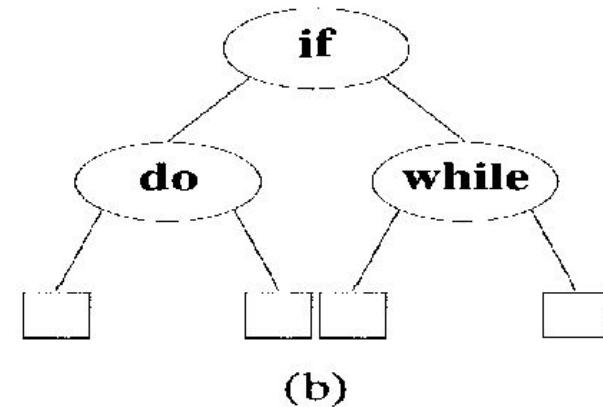
$$nC_r = n! / (r! * (n - r)!)$$

When $n=3$, no. of BST trees are $\frac{1}{4}((2^3)C_3) = 5$

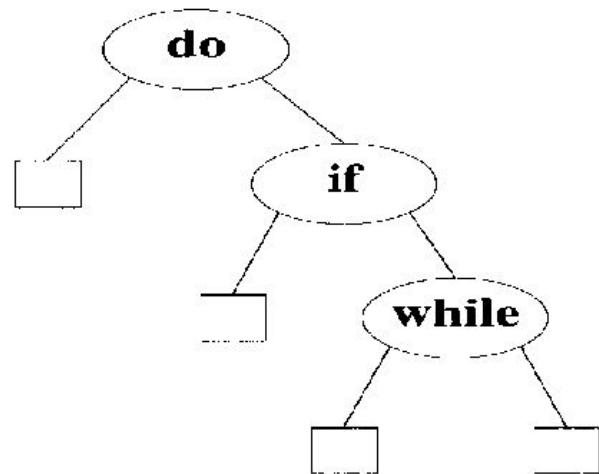
Different possible BST with n=3



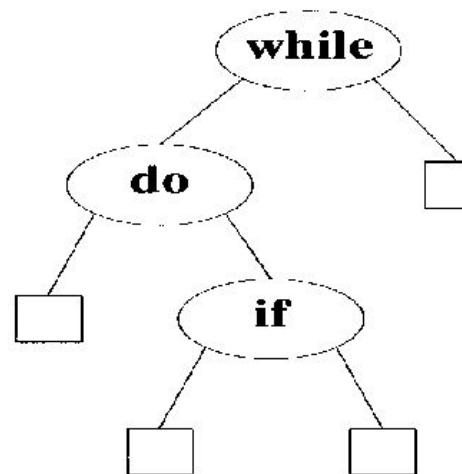
(a)



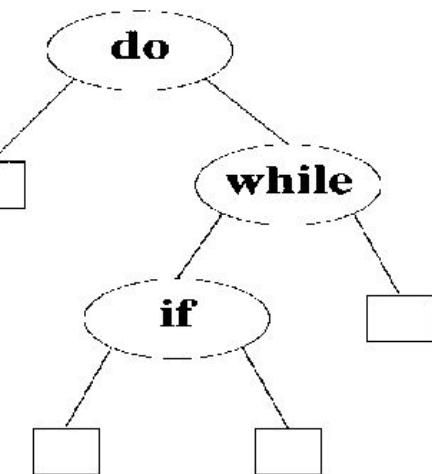
(b)



(c)



(d)

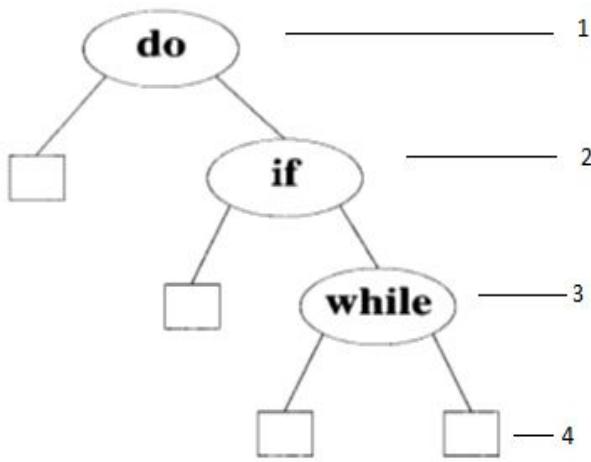


(e)

Example

With equal probabilities $p(i) = q(i) = 1/7$ for all i , find the optimal binary search tree.

$$\sum_{i=1}^n P_i * \text{level}(a_i) + \sum_{i=0}^n Q_i * (\text{level}(E_i) - 1)$$



Cost of **successful** search is :

$$1/7*1 + 1/7*2 + 1/7*3 = 6/7$$

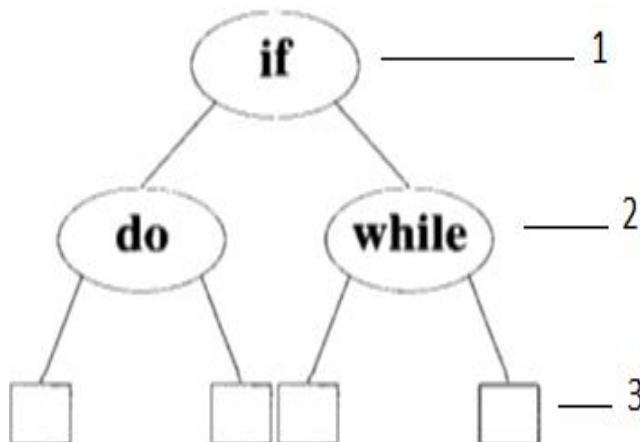
Cost of **unsuccessful** search is :

$$1/7*(2-1) + 1/7*(3-1) +$$

$$1/7*(4-1) + 1/7*(4-1) = 9/7$$

$$\begin{array}{ccccccc} & & & & & & \\ | & & | & & | & & | \\ E_0 & & E_1 & & E_2 & & E_3 \end{array}$$

$$\text{The cost is } 6/7 + 9/7 = 15/7$$



Cost of **successful** search is :

$$1/7*1 + 1/7*2 + 2/7*3 = 5/7$$

Cost of **unsuccessful** search is :

$$1/7*(3-1) + 1/7*(3-1) +$$

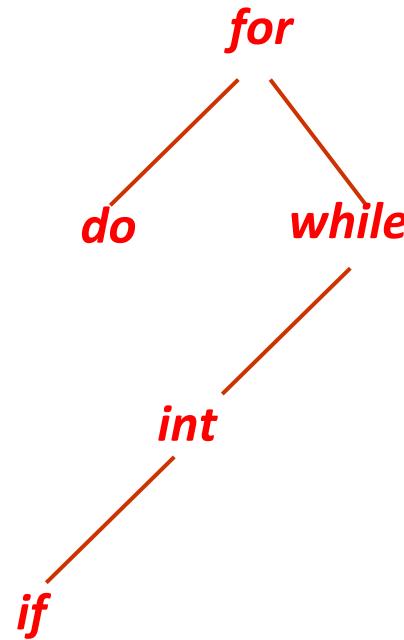
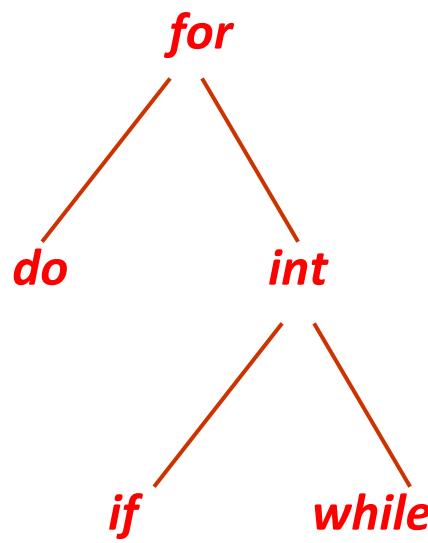
$$1/7*(3-1) + 1/7*(2-1) = 8/7$$

$$\begin{array}{ccccccc} & & & & & & \\ | & & | & & | & & | \\ E_0 & & E_1 & & E_2 & & E_3 \end{array}$$

$$\text{The cost is } 5/7 + 8/7 = 13/7$$

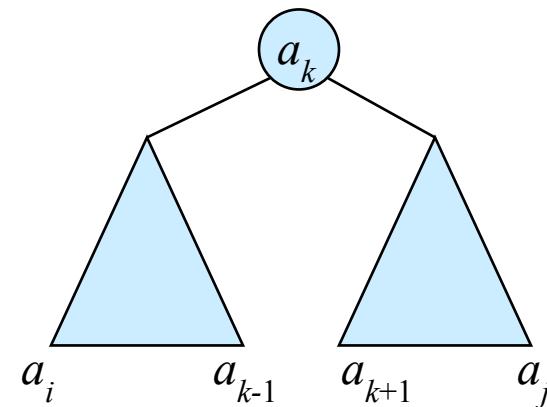
Exercise

- Given the set of identifiers $(a_1, a_2, a_3, a_4, a_5) = (\text{do}, \text{for}, \text{int}, \text{if}, \text{while})$ with probabilities of successful search is $(2, 3, 2, 2, 3)$ and probabilities of unsuccessful search $(1, 2, 1, 2, 2, 1)$ for convenience the probabilities are multiplied by 21



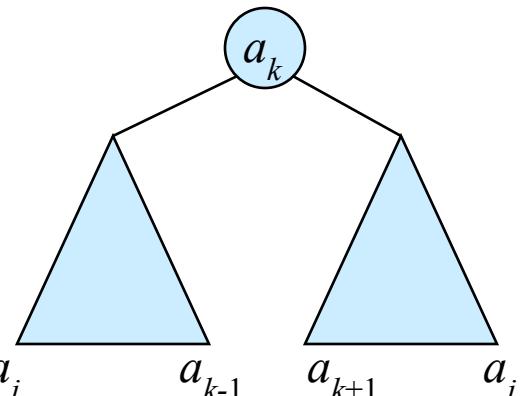
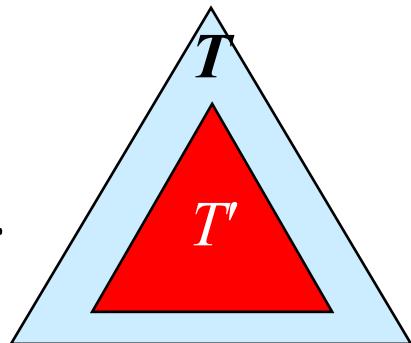
Dynamic Approach to OBST

- To apply dynamic programming, the construction of tree is viewed as the result of a sequence of decisions and the principle of optimality holds when applied to the problem state resulting from a decision.
- A possible approach to this would be to make a decision as to which of the a_i 's should be assigned to the root node of the tree.
- To find an optimal BST:
 - Examine all candidate roots a_k , for $i \leq k \leq j$
 - Determine all optimal BSTs containing a_i, \dots, a_{k-1} and containing a_{k+1}, \dots, a_j



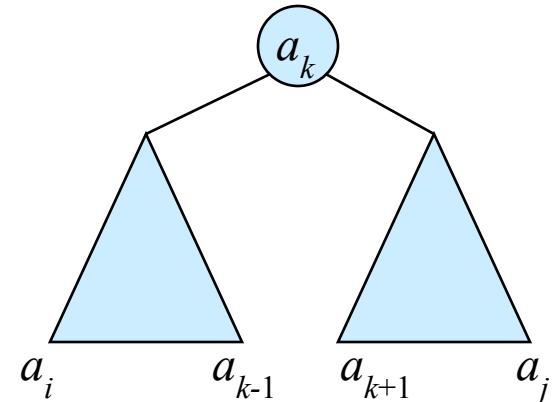
Optimal Substructure

- Any subtree of a BST contains keys in a contiguous range a_i, \dots, a_j for some $1 \leq i \leq j \leq n$.
- If T is an optimal BST and T contains subtree T' with keys a_i, \dots, a_j , then T' must be an optimal BST for keys a_i, \dots, a_j .
- One of the keys in a_i, \dots, a_j , say a_k , where $i \leq k \leq j$, must be the root of an optimal subtree for these keys.
- Left subtree of a_k contains a_i, \dots, a_{k-1} .
- Right subtree of a_k contains a_{k+1}, \dots, a_j .
- To find an optimal BST:
 - Examine all candidate roots a_k , for $i \leq k \leq j$
 - Determine all optimal BSTs containing a_i, \dots, a_{k-1} and containing a_{k+1}, \dots, a_j



Recursive Solution

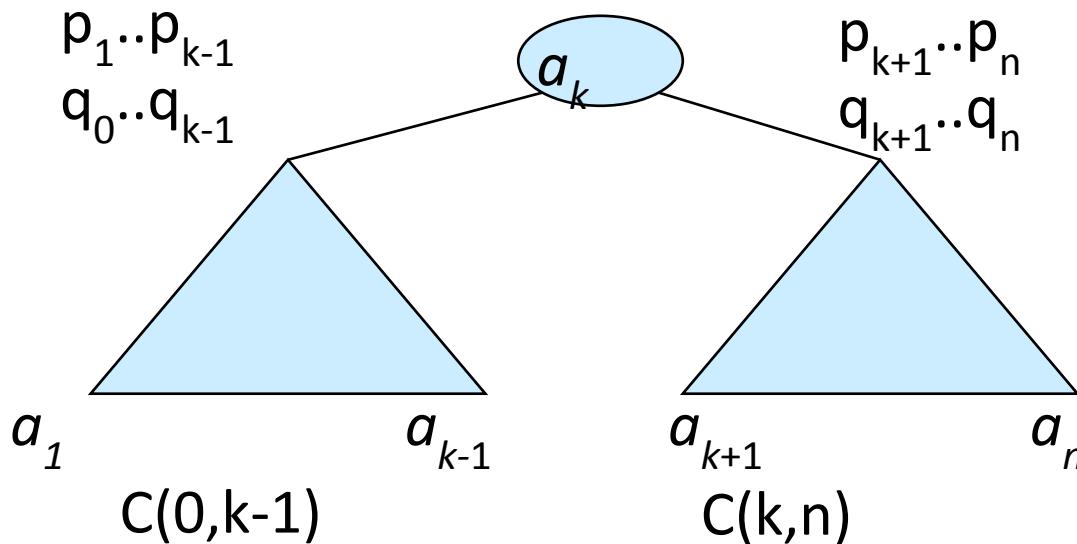
- Find optimal BST for a_i, \dots, a_j , where $1 \leq i < j \leq n, j \geq i-1$.
When $j = i-1$, the tree is empty.
- Define $c[i, j] =$ expected search cost of optimal BST for a_i, \dots, a_j .
- If $j = i$, then $c[i, j] = 0$.
- If $j \geq i$,
 - Select a root a_k , for some $i < k \leq j$.
 - Recursively make an optimal BSTs
 - for a_i, \dots, a_{k-1} in the left subtree, and
 - for a_{k+1}, \dots, a_j in the right subtree.



Recursive Solution

- Find optimal BST for a_i, \dots, a_j ,
- Let $C(i, j)$ denote the cost of an optimal binary search tree containing a_i, \dots, a_j .
- The cost of the optimal binary search tree with a_k as its root :

$$C(i,j) = \min_{i < k \leq j} \left\{ P_k + \left[Q_{i-1} + \sum_{i=1}^{k-1} (P_i + Q_i) + C(i, k-1) \right] + \left[Q_k + \sum_{i=k+1}^j (P_i + Q_i) + C(k, j) \right] \right\}$$



Dynamic Approach

- To apply dynamic programming, the construction of tree is **viewed as the result of a sequence of decisions** and the principle of optimality holds when applied to the problem state resulting from a decision.
- A possible approach to this would be to make a decision as **to which of the a_i 's should be assigned to the root node of the tree.**
- Dynamic programming in solving OBST problem uses the following formulas:
 - $w(i, j)$, $r(i, j)$, $c(i, j)$

OBST

$$W(i, j) = q(i) + \sum_{m=i+1}^j (q(m) + p(m))$$

Can be written as $\mathbf{W(i, j)} = \mathbf{p(j)} + \mathbf{q(j)} + \mathbf{w(i, j-1)}$

Where P_j and Q_j are the probabilities of left and right sub-trees. When an element in level d is searched $d-1$ elements are compared, hence its cost is included

$$c(i, j) = \min_{i < k \leq j} \{ c(i, k - 1) + c(k, j) + p(k) + w(i, k - 1) + w(k, j) \}$$

can be written as $\mathbf{c(i, j)} = \min_{i < k \leq j} \{ \mathbf{c(i, k - 1)} + \mathbf{c(k, j)} \} + \mathbf{w(i, j)}$

Let $R(i, j)$ be the value of k that minimizes the above equ.

$C(i, i) = 0$ and $W(i, i) = Q_i$ where $0 \leq i \leq n$

Compute all $c(i, j)$ with $j-i=1, j-i=2$ and so on to $j-i=n$

Example

- Let $n=4$, and $(a_1, a_2, a_3, a_4) = (\text{do}, \text{if}, \text{int}, \text{while})$ with probabilities of successful search is $(3, 3, 1, 1)$ and probabilities of unsuccessful search $(2, 3, 1, 1, 1)$ for convenience the probabilities are multiplied by 16
- Initially
 - $W(i,i)=q(i)$, $c(i,i) = 0$ and $r(i,i)=0$
 - Compute all $c(i, j)$ with $j-i=1, j-i=2$ and so on upto $j-i=n$

$$w(0, 1) = p(1) + q(1) + w(0, 0) = 8$$

$$c(0, 1) = w(0, 1) + \min\{c(0, 0) + c(1, 1)\} = 8$$

$$r(0, 1) = 1$$

$$w(1, 2) = p(2) + q(2) + w(1, 1) = 7$$

$$c(1, 2) = w(1, 2) + \min\{c(1, 1) + c(2, 2)\} = 7$$

$$r(0, 2) = 2$$

Computation of OBST

	0	1	2	3	4
0	$w_{00} = 2$ $c_{00} = 0$ $r_{00} = 0$	$w_{11} = 3$ $c_{11} = 0$ $r_{11} = 0$	$w_{22} = 1$ $c_{22} = 0$ $r_{22} = 0$	$w_{33} = 1$ $c_{33} = 0$ $r_{33} = 0$	$w_{44} = 1$ $c_{44} = 0$ $r_{44} = 0$
1	$w_{01} = 8$ $c_{01} = 8$ $r_{01} = 1$	$w_{12} = 7$ $c_{12} = 7$ $r_{12} = 2$	$w_{23} = 3$ $c_{23} = 3$ $r_{23} = 3$	$w_{34} = 3$ $c_{34} = 3$ $r_{34} = 4$	
2	$w_{02} = 12$ $c_{02} = 19$ $r_{02} = 1$	$w_{13} = 9$ $c_{13} = 12$ $r_{13} = 2$	$w_{24} = 5$ $c_{24} = 8$ $r_{24} = 3$		
3	$w_{03} = 14$ $c_{03} = 25$ $r_{03} = 2$	$w_{14} = 11$ $c_{14} = 19$ $r_{14} = 2$			
4	$w_{04} = 16$ $c_{04} = 32$ $r_{04} = 2$				

Problem

- Construct the OBST for the following set (a_1, a_2, a_3, a_4) with $p(1) = 1/20$, $p(2)=1/5$, $p(3)= 1/10$, $p(4)=1/20$ and $q(0)=1/5$, $q(1)=1/10$, $q(2)=1/5$, $q(3)=1/20$, $q(4)=1/20$.

```

1  Algorithm OBST( $p, q, n$ )
2  // Given  $n$  distinct identifiers  $a_1 < a_2 < \dots < a_n$  and probabilities
3  //  $p[i]$ ,  $1 \leq i \leq n$ , and  $q[i]$ ,  $0 \leq i \leq n$ , this algorithm computes
4  // the cost  $c[i, j]$  of optimal binary search trees  $t_{ij}$  for identifiers
5  //  $a_{i+1}, \dots, a_j$ . It also computes  $r[i, j]$ , the root of  $t_{ij}$ .
6  //  $w[i, j]$  is the weight of  $t_{ij}$ .
7  {
8      for  $i := 0$  to  $n - 1$  do
9      {
10         // Initialize.
11          $w[i, i] := q[i]; r[i, i] := 0; c[i, i] := 0.0;$ 
12         // Optimal trees with one node
13          $w[i, i + 1] := q[i] + q[i + 1] + p[i + 1];$ 
14          $r[i, i + 1] := i + 1;$ 
15          $c[i, i + 1] := q[i] + q[i + 1] + p[i + 1];$ 
16     }
17      $w[n, n] := q[n]; r[n, n] := 0; c[n, n] := 0.0;$ 

```

```

18   for  $m := 2$  to  $n$  do // Find optimal trees with  $m$  nodes.
19     for  $i := 0$  to  $n - m$  do
20     {
21        $j := i + m;$ 
22        $w[i, j] := w[i, j - 1] + p[j] + q[j];$ 
23       // Solve 5.12 using Knuth's result.
24        $k := \text{Find}(c, r, i, j);$ 
25       // A value of  $l$  in the range  $r[i, j - 1] \leq l$ 
26       //  $\leq r[i + 1, j]$  that minimizes  $c[i, l - 1] + c[l, j];$ 
27        $c[i, j] := w[i, j] + c[i, k - 1] + c[k, j];$ 
28        $r[i, j] := k;$ 
29     }
30   write ( $c[0, n]$ ,  $w[0, n]$ ,  $r[0, n]$ );
31 }

```

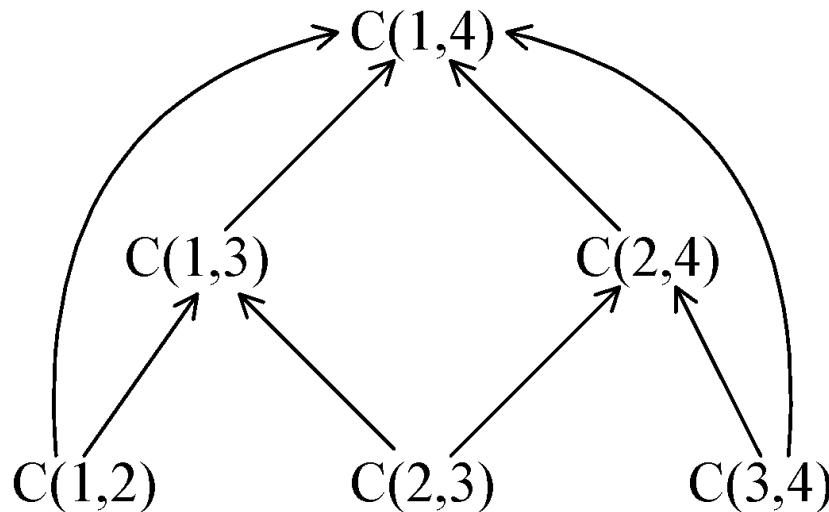
```

1 Algorithm  $\text{Find}(c, r, i, j)$ 
2 {
3    $min := \infty;$ 
4   for  $m := r[i, j - 1]$  to  $r[i + 1, j]$  do
5     if  $(c[i, m - 1] + c[m, j]) < min$  then
6     {
7        $min := c[i, m - 1] + c[m, j]; l := m;$ 
8     }
9   return  $l;$ 
10 }

```

Computation relationships of subtrees

- e.g. $n=4$



- Time complexity : $O(n^3)$
when $j-i=m$, there are $(n-m)$ $C(i, j)$'s to compute.
Each $C(i, j)$ with $j-i=m$ can be computed in $O(m)$ time.

$$O\left(\sum_{1 \leq m \leq n} m(n - m)\right) = O(n^3)$$

0/1 Knapsack problem

Given a knapsack or (bag) with maximum capacity M , and a set of n items. Each item i has weight w_i and Profit p_i

If a fraction x_i , $0 \leq x_i \leq 1$, of item i is placed into the knapsack, then a profit of $p_i x_i$ is earned

Item #	Weight	Profit
1	1	8
2	3	6
3	5	5

The Problem of 0/1 Knapsack is to select the items into the Knapsack, not exceeding the capacity (M), with an objective to maximize the total profit earned.

$$\max \sum_{1 \leq i \leq n} p_i x_i$$

$$\text{subject to } \sum_{1 \leq i \leq n} w_i \leq M$$

where $x_i = 0$ or 1 , $1 \leq i \leq n$

0/1 Knapsack

- A solution to knapsack problem can be viewed as making sequence of decisions on x_1, x_2, \dots, x_n .
- A decision on x_i involves which of the values 0 or 1 is to be assigned to it.
- A decision on x_n will make us be in one of two states
 - The capacity remaining same and no profit accrued
 - The capacity remaining is $m-w_n$ and profit p_n is accrued
- Let $f_j(y)$ be the optimal solution to knap(1,j,y)
 - $f_n(m) = \max\{f_{n-1}(m), f_{n-1}(m-w_n)+p_n\}$

0/1 knapsack order pair method

$$f_i(y) = \max\{f_{i-1}(y), f_{i-1}(y-w_i)+p_i\}$$

We use ordered set $S^i = \{(f(y_j), y_j) | 1 \leq j \leq n\}$ to represent $f_i(y)$.

Each member of S^i is an ordered pair (P, W) , where $P = f_i(y_j)$ and $W = y_j$.

Note $S^0 = \{(0, 0)\}$, we can compute S^{i+1} from S^i

$S^i = \{(P, W)\}$ be the order pairs after inclusion of i^{th} item
..... (1)

To compute S^{i+1} first compute S_{i+1}^i

$S_{i+1}^i = \{(P, W) | (P' + p_{i+1}, W' + w_{i+1}) (P', W') \in S^i\}$
...(2)

S^{i+1} can be computed by merging the pairs S^i and S_{i+1}^i

$S^{i+1} = S^i \cup S_{i+1}^i$
..... (3)

Dominance / Purge rule

- If s^{i+1} contains 2 pairs (p_j, w_j) & (p_k, w_k) with the property that $p_j \leq p_k$ & $w_j \geq w_k$ then the pair (p_j, w_j) can be discarded known as dominance rule.
 - i.e.- Arrange all the pairs in s^{i+1} increasing order of weights. If weights increases then profit should also increase, if not discard the pair.

Optimal Solution

- If (P_1, W_1) is the last tuple in s^n , a set of 0/1 values for x_i 's can be determined carrying out a search through the S^i 's.
- Let $x_n = 0$ if $(P_1, W_1) \in S^{n-1}$
- if $(P_1, W_1) \notin S^{n-1}$ then $(P_1 - p_n, W_1 - w_n) \in S^{n-1}$ and we can set $x_n = 1$
- This leaves us determine how (P_1, W_1) and $(P_1 - p_n, W_1 - w_n)$ obtained in S^{n-1}
- This can be done recursively.

Example

Let $n=3$, $(w_1, w_2, w_3) = (2, 3, 4)$, $(p_1, p_2, p_3) = (1, 2, 5)$ and $m=6$

Solution :

$$S^i = \{(P, W)\}$$

$$S^0 = \{(0, 0)\} \text{-----initially no objects}$$

$$S^0_1 = \{(1, 2)\} \text{----- inclusion of item 1}$$

$$S^{i+1} = S^i \cup S^i_1$$

$$S^1 = S^0 \cup S^0_1 \text{ Therefore } S^1 = \{(0, 0), (1, 2)\}$$

$$S^1_1 = \{(2, 3), (3, 5)\} \quad S^2 = \{(0, 0), (1, 2), (2, 3), (3, 5)\}$$

$$S^2_1 = \{(5, 4), (6, 6), \cancel{(7, 7)}, \cancel{(8, 9)}\} \text{ as weight} > M$$

$$S^3 = \{(0, 0), (1, 2), (2, 3), (5, 4), (6, 6)\}$$

Note the pair $(3, 5)$ in S^2 is discarded by dominance rule between $(3, 5)$ $(5, 4)$

Constructing Optimal Solution

Item	Order Pairs	search	X _i values
0	$S^0 = \{(0,0)\}$	---	---
1	$S^1 = \{(0,0), (1,2)\}$	(1,2)	$X_1 = 1$ this order pair is in s^1 and not in s^0
2	$S^2 = \{(0,0), (1,2), (2,3), (3,5)\}$	(6-5, 6-4) is (1,2)	$X_2 = 0$ this order pair is in s^2 in s^1
3	$S^3 = \{(0,0)(1,2)(2,3),(5,4),(6,6)\}$	(6,6) max profit	$X_3 = 1$ this order pair is only in s^3

Algorithm

```
1  Algorithm DKP( $p, w, n, m$ )
2  {
3       $S^0 := \{(0, 0)\}$ ;
4      for  $i := 1$  to  $n - 1$  do
5      {
6           $S_1^{i-1} := \{(P, W) | (P - p_i, W - w_i) \in S^{i-1} \text{ and } W \leq m\}$ ;
7           $S^i := \text{MergePurge}(S^{i-1}, S_1^{i-1})$ ;
8      }
9      ( $PX, WX$ ) := last pair in  $S^{n-1}$ ;
10     ( $PY, WY$ ) :=  $(P' + p_n, W' + w_n)$  where  $W'$  is the largest  $W$  in
11         any pair in  $S^{n-1}$  such that  $W + w_n \leq m$ ;
12     // Trace back for  $x_n, x_{n-1}, \dots, x_1$ .
13     if ( $PX > PY$ ) then  $x_n := 0$ ;
14     else  $x_n := 1$ ;
15     TraceBackFor( $x_{n-1}, \dots, x_1$ );
16 }
```

Analysis of 0/1 Knapsack

- The time needed to generate S^i from S^{i-1} is $\Theta(|S^{i-1}|)$
- The time needed to compute all the S^i 's, $0 \leq i < n$, is $\Theta(\sum |S^{i-1}|)$
 - $1 + 2 + 2^2 + \dots + 2^{n-1} = \frac{2^n - 1}{2 - 1} = 2^n - 1$
- Since $|S^i| < 2^i$
- The time needed to compute all S^i 's is $O(2^n)$

All Pairs Shortest Path

- Let $G = (V, E)$ be a directed graph with n vertices.
- Let cost be a cost adjacency matrix for G such that
 - $\text{cost}(i, i) = 0, 1 \leq i \leq n$.
 - $\text{cost}(i, j)$ is the length (or cost) of edge (i, j) if $(i, j) \in E(G)$ and
 - $\text{cost}(i, j) = \infty$ if $i \neq j$ and $(i, j) \notin E(G)$.

The all-pairs shortest-path problem is to determine a matrix A such that $A(i, j)$ is the length of a shortest path from i to j .

Restriction : G has no edges with negative cycles.

Note : If we allow G to contain a cycle of negative length, then the shortest path between any two vertices on this cycle has length = $-\infty$

All Pairs Shortest Path

Simplest shortest path from i to j is a direct edge (i, j)

Let i to j be the shortest path originating from i , going through vertex k and terminating at j .

General case: $i, v_1, v_2, \dots, V_k, \dots, v_m, j$

All of $\{v_1, v_2, v_3, \dots, v_m\}$ are distinct, and different from i & j

- Then sub-paths i to k and k to j must be shortest path from $i-k$ and $k-j$. (by Principle of Optimality)
- If k is the intermediate vertex with highest index, then the $i-k$ and $k-j$ paths are shortest $i-k$ and $k-j$ paths respectively in G going through no vertex with index $> k-1$.
- Let $A^k(i, j) :$ represent the length of a shortest path from $i - j$ going through no vertex of index $> k$.

All Pairs Shortest path

From $A^{k-1}(i, j)$ to $A^k(i, j)$

Case 1: Shortest path via $\{1, 2, \dots, k\}$ **not going** thru vertex k

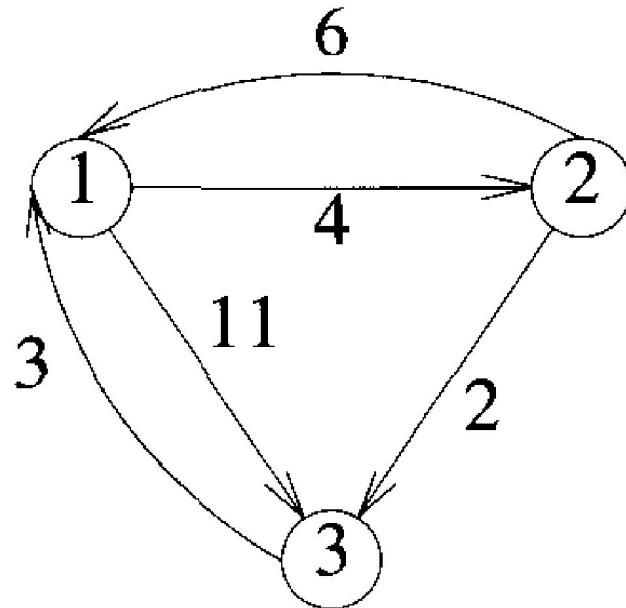
$$A^k(i, j) = A^{k-1}(i, j)$$

Case 2: Shortest path via $\{1, 2, \dots, k\}$ going **thru** vertex k

$$A^k(i, j) = A^{k-1}(i, k) + A^{k-1}(k, j)$$

$$A^k(i, j) = \min_{1 \leq k \leq n} \{(A^{k-1}(i, j), A^{k-1}(i, k) + A^{k-1}(k, j))\}$$

APSP Example



Compute A^0 (i,j) for
 $i=1$ to 3, $j=1$ to 3

$$\text{cost}(1,1)=0 \quad \text{cost}(1,2)=4$$

$$\text{cost}(1,3)=11$$

$$\text{cost}(2,1)=6 \quad \text{cost}(2,2)=0$$

$$\text{cost}(3,1)=3 \quad \text{cost}(3,2)=\infty$$

$$\text{cost}(i, i) = 0, 1 \leq i \leq n.$$

$\text{cost}(i, j)$ is the length (or cost) of edge (i, j) if $(i, j) \in E(G)$ and

$$\text{cost}(i, j)=\infty \text{ if } i \neq j \text{ and } (i, j) \notin E(G)$$

A^0	1	2	3
1	0	4	11
2	6	0	2
3	3	∞	0

(b) A^0

$$\text{cost}(2,3)=2$$

$$\text{cost}(3,3)=0$$

ASAP

A^1	1	2	3
1	0	4	11
2	6	0	2
3	3	7	0

Compute $A^k(i,j) = \min \{ A^{k-1}(i,j), A^{k-1}(i,k) + A^{k-1}(k,j) \}$

For $k=1$ to 3, $i=1$ to 3 and $j=1$ to 3

$K=1$, $i=1$ to 3 & $j=1$ to 3

(c) A^1

$$A^1(1,1) = \min \{ A^0(1,1), A^0(1,1) + A^0(1,1) \} = \min \{ 0, 0+0 \} = 0$$

$$A^1(1,2) = \min \{ A^0(1,2), A^0(1,1) + A^0(1,2) \} = \min \{ 4, 0+4 \} = 4$$

$$A^1(1,3) = \min \{ A^0(1,3), A^0(1,1) + A^0(1,3) \} = \min \{ 0, 0+11 \} = 11$$

$$A^1(2,1) = \min \{ A^0(2,1), A^0(2,1) + A^0(1,1) \} = \min \{ 6, 6+0 \} = 6$$

$$A^1(2,2) = \min \{ A^0(2,2), A^0(2,1) + A^0(1,2) \} = \min \{ 0, 6+4 \} = 0$$

$$A^1(2,3) = \min \{ A^0(2,3), A^0(2,1) + A^0(1,3) \} = \min \{ 2, 6+3 \} = 2$$

$$A^1(3,1) = \min \{ A^0(3,1), A^0(3,1) + A^0(1,1) \} = \min \{ 3, 3+0 \} = 3$$

$$A^1(3,2) = \min \{ A^0(3,2), A^0(3,1) + A^0(1,2) \} = \min \{ \infty, 3+4 \} = 7$$

$$A^1(3,3) = \min \{ A^0(3,3), A^0(3,1) + A^0(1,3) \} = \min \{ 0, 3+11 \} = 0$$

Conti..

- Compute $A^k(i,j) = \min \{ A^{k-1}(i,j), A^{k-1}(i,k) + A^{k-1}(k,j) \}$
- Now $k=2$

$$A^2(1,1) = \min \{ A^1(1,1), A^1(1,2) + A^1(2,1) \} = \min \{ 0, 4+5 \} = 0$$

$$A^2(1,2) = \min \{ A^1(1,2), A^1(1,2) + A^1(2,2) \} = \min \{ 4, 4+0 \} = 4$$

$$A^2(1,3) = \min \{ A^1(1,3), A^1(1,2) + A^0(2,3) \} = \min \{ 11, 4+2 \} = 6$$

$$A^2(2,1) = \min \{ A^1(2,1), A^1(2,2) + A^1(1,1) \} = \min \{ 6, 0+6 \} = 6$$

$$A^2(2,2) = \min \{ A^1(2,2), A^1(2,2) + A^1(2,2) \} = \min \{ 0, 0+0 \} = 0$$

$$A^2(2,3) = \min \{ A^1(2,3), A^1(2,2) + A^1(2,3) \} = \min \{ 2, 0+2 \} = 2$$

$$A^2(3,1) = \min \{ A^1(3,1), A^1(3,2) + A^1(2,1) \} = \min \{ 3, 3+6 \} = 3$$

$$A^2(3,2) = \min \{ A^1(3,2), A^1(3,2) + A^1(2,2) \} = \min \{ 7, 7+0 \} = 7$$

$$A^2(3,3) = \min \{ A^1(3,3), A^1(3,2) + A^1(2,3) \} = \min \{ 0, 7+2 \} = 0$$

A^2	1	2	3
1	0	4	6
2	6	0	2
3	3	7	0
(d) A^2			

Conti..

- Compute $A^k(i,j) = \min \{ A^{k-1}(i,j), A^{k-1}(i,k) + A^{k-1}(k,j) \}$
- Now $k=3$

$$A^3(1,1) = \min \{ A^2(1,1), A^2(1,3) + A^2(3,1) \} = \min \{ 0, 6+3 \} = 0$$

$$A^3(1,2) = \min \{ A^2(1,2), A^2(1,3) + A^2(3,2) \} = \min \{ 4, 6+7 \} = 4$$

$$A^3(1,3) = \min \{ A^2(1,3), A^2(1,3) + A^2(3,3) \} = \min \{ 11, 6+0 \} = 6$$

$$A^3(2,1) = \min \{ A^2(2,1), A^2(2,3) + A^2(3,1) \} = \min \{ 6, 2+3 \} = 5$$

$$A^3(2,2) = \min \{ A^2(2,2), A^2(2,3) + A^2(3,2) \} = \min \{ 0, 2+7 \} = 0$$

$$A^3(2,3) = \min \{ A^2(2,3), A^2(2,3) + A^2(3,3) \} = \min \{ 2, 2+0 \} = 2$$

$$A^3(3,1) = \min \{ A^2(3,1), A^2(3,3) + A^2(3,1) \} = \min \{ 3, 0+3 \} = 3$$

$$A^3(3,2) = \min \{ A^2(3,2), A^2(3,3) + A^2(3,2) \} = \min \{ 7, 0+7 \} = 7$$

$$A^3(3,3) = \min \{ A^2(3,3), A^2(3,3) + A^2(3,3) \} = \min \{ 0, 0+0 \} = 0$$

A^3	1	2	3
1	0	4	6
2	5	0	2
3	3	7	0

(e) A^3

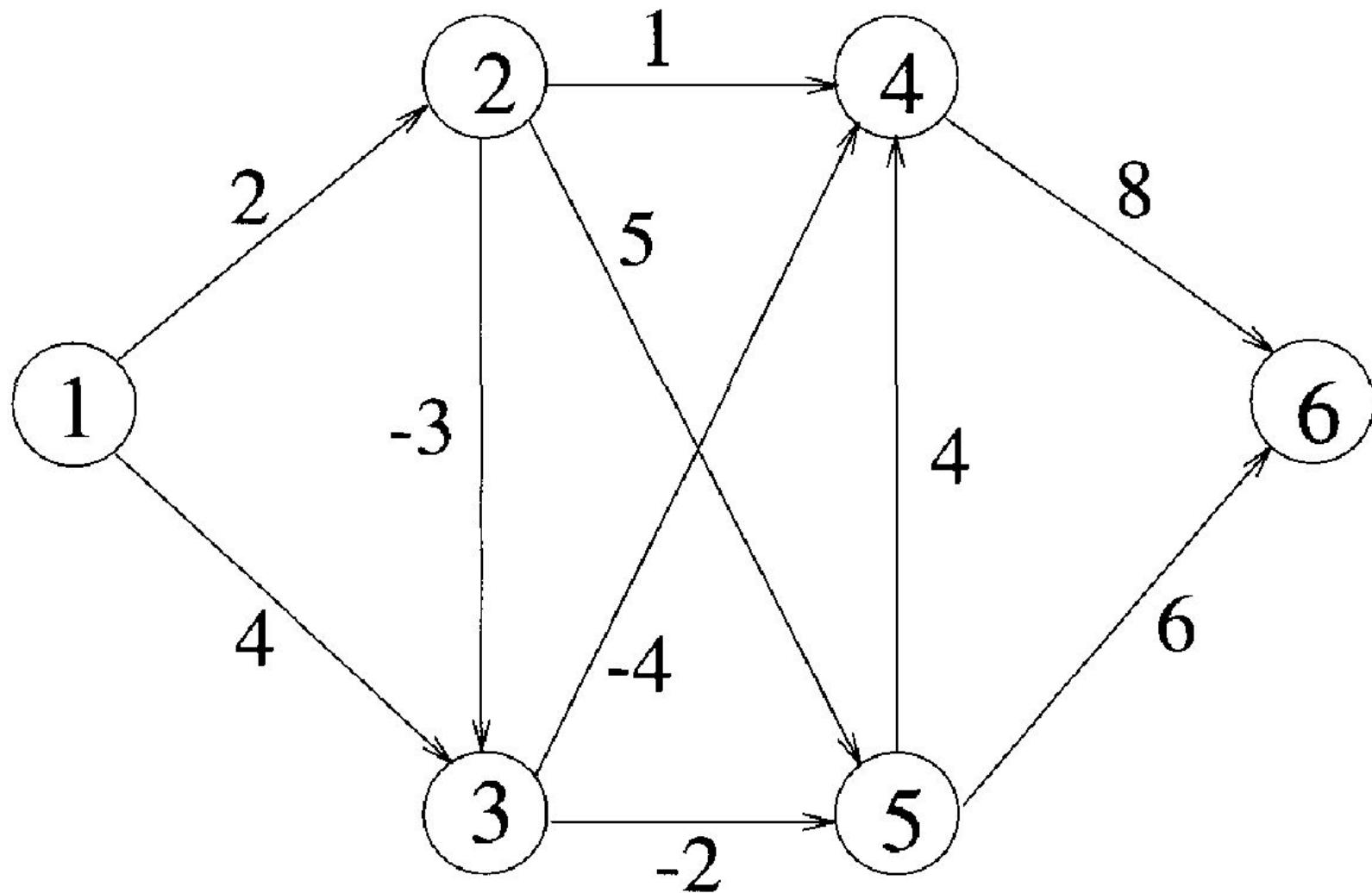
APSP Algorithm

```
0  Algorithm AllPaths(cost, A, n)
1    // cost[1 : n, 1 : n] is the cost adjacency matrix of a graph with
2    // n vertices; A[i, j] is the cost of a shortest path from vertex
3    // i to vertex j. cost[i, i] = 0.0, for 1 ≤ i ≤ n.
4    {
5        for i := 1 to n do
6            for j := 1 to n do
7                A[i, j] := cost[i, j]; // Copy cost into A.
8            for k := 1 to n do
9                for i := 1 to n do
10                   for j := 1 to n do
11                       A[i, j] := min(A[i, j], A[i, k] + A[k, j]);
12    }
```

Time Complexity of APSP

- The no. of Matrices Computed are n .
- Each matrix generation requires n^2 entries.
- ie., the number of $A^k(i, j)$ computations are n^2 .
- The no of A^k required are n .
- Total time is $O(n^3)$

Exercise – All Pair Shortest Path



Travelling Sales Person Problem

- A tour of G is a directed simple cycle that includes every vertex in V . The cost of a tour is the sum of the cost of the edges on the tour. The Travelling sales person problem is to find a tour of minimum cost.
- **Applications**
 - Postal van to pick up mail from mail boxes located at n different sites.
 - Robot arm to tighten the nuts on some piece of machinery on an assembly line.

Solving TSP

- Regard a tour to be a simple path that starts and end at vertex 1.
- Every tour consists of an edge $\langle 1, k \rangle$ for some $k \in V - \{1\}$ and a path from k to vertex 1. The path from vertex k to vertex 1 goes through each vertex in $V - \{1, k\}$ exactly once.
- Let $g(i, S)$ be the length of a shortest path starting at vertex i , going through all vertices in S and terminating at vertex 1.
- $g(1, V - \{1\})$ is the length of an optimal salesperson tour.
- From the principle of optimality it follows that:

$$g(1, V - \{1\}) = \min_{2 \leq k \leq n} \{c_{1k} + g(k, V - \{1, k\})\} \quad \dots\dots(1)$$

- Generalizing from (1) we obtain (for $i \notin S$)
$$g(i, S) = \min_{j \in S} \{c_{ij} + g(j, S - \{j\})\} \quad \dots\dots\dots (2)$$
 - Formula (2) may be solved for $g(1, V - \{1\})$ if we know $g(k, V - \{1, k\})$ for all choices of k .
 - The g values may be obtained by using (2).
 - Clearly, $g(i, \emptyset) = c_{i,1}$, $1 \leq i \leq n$.
 - Hence, we may use (2) to obtain $g(i, S)$ for all S of size 1.
 - Then we can obtain $g(i, S)$ for S which $|S| = 2$.
 - Then, we can obtain $g(i, S)$ for S which $|S| = 3$, etc.
 - When $|S| < n - 1$, the value of i and S for which $g(i, S)$ is needed are such that $i \neq 1$; $1 \notin S$ and $i \notin S$.

Example

- Consider the directed graph which has the adjacency matrix as follows:

$$\begin{matrix} 0 & 10 & 15 & 20 \\ 5 & 0 & 9 & 10 \\ 6 & 13 & 0 & 12 \\ 8 & 8 & 9 & 0 \end{matrix}$$

$$g(2, \emptyset) = c_{21} = 5 ; g(3, \emptyset) = c_{31} = 6; g(4, \emptyset) = c_{41} = 8.$$

Using (2) we obtain

$$g(2, \{3\}) = c_{23} + g(3, \emptyset) = 9 + 6 = 15$$

$$g(2, \{4\}) = c_{24} + g(4, \emptyset) = 10 + 8 = 18$$

$$g(3, \{2\}) = c_{32} + g(2, \emptyset) = 13 + 5 = 18$$

$$g(3, \{4\}) = c_{34} + g(4, \emptyset) = 12 + 8 = 20$$

$$g(4, \{2\}) = c_{42} + g(2, \emptyset) = 8 + 5 = 13$$

$$g(4, \{3\}) = c_{43} + g(3, \emptyset) = 9 + 6 = 15$$

- Next, we compute $g(i, S)$ with $|S| = 2$ and $i \neq 1; 1 \notin S$ and $i \in S$.

$$g(2, \{3,4\}) = \min \{c_{23} + g(3, \{4\}), c_{24} + g(4, \{3\})\} = 25$$

$$g(3, \{2,4\}) = \min \{c_{32} + g(2, \{4\}), c_{34} + g(4, \{2\})\} = 25$$

$$g(4, \{2,3\}) = \min \{c_{42} + g(2, \{3\}), c_{43} + g(3, \{2\})\} = 23$$

- Finally, from (2) we obtain:

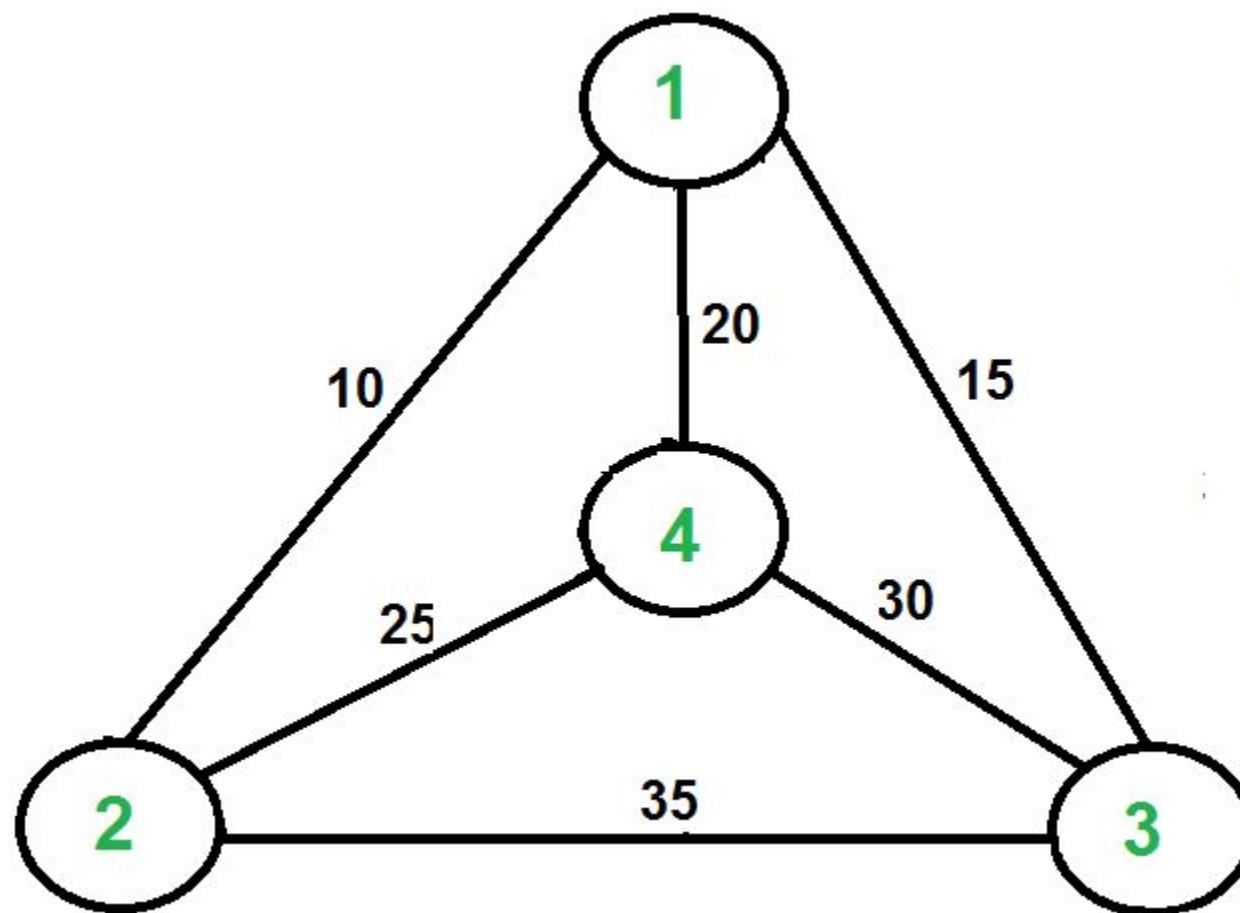
$$g(1, \{2, 3, 4\}) = \min \{c_{12} + g(2, \{3, 4\}), c_{13} + g(3, \{2, 4\}), c_{14} + g(4, \{2, 3\})\} = \min \{35, 40, 43\} = 35$$

- An optimal tour of the graph has length 35.
- A tour of this length may be constructed if we retain with each $g(i, S)$ the value of j that minimizes the right hand side of (2). Let $J(i, S)$ be this value.
- $J(1, \{2, 3, 4\}) = 2$. Thus the tour starts from 1 and goes to 2. The remaining tour can be obtained from $g(2, \{3, 4\})$. $J(2, \{3, 4\}) = 4$. Thus the next edge is $\langle 2, 4 \rangle$. The remaining tour is for $g(4, \{3\})$. $J(4, \{3\}) = 3$. The optimal tour is $\langle 1, 2, 4, 3, 1 \rangle$

Complexity of the method

- Let N be the number of $g(i, S)$ that have to be computed before (1) may be used to compute $g(1, V - \{1\})$. For each value of $|S|$ there are $n - 1$ choices for i . The number of distinct sets of sets S of size k not including 1 and i is
 - C_{n-2}^k
- Hence $N = \sum_{k=0}^{n-2} (n - 1) C_{n-2}^k = (n - 1)2^{n-2}$
- $\frac{n}{0}$
- An algorithm that proceeds to find an optimal tour by making use of (1) and (2) will require $O(n^2 2^n)$ time since the computation of $g(i, S)$ with $|S| = k$ requires $k - 1$ comparisons when solving (2).
- It's better than enumerating all $n!$ different tours to find the best one.
- The most serious drawback of this dynamic programming solution is the space needed. The space needed is $O(n2^n)$. This is too large even for modest values of n .

Try for this



Solve the TSP

0	11	10	9	6
8	0	7	3	4
8	4	0	4	8
11	10	5	0	5
6	9	5	5	0

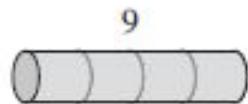
Optimal Rod Cutting Problem

- **Problem:** Find best way to cut a rod of length n
 - Given: rod of *length n*
 - Assume that each length rod has *a price* p_i
 - Find best set of cuts to get maximum revenue (ie r_n)
 - Each cut is integer length
 - Can use any number of cuts, from 0 to $n-1$
 - No cost for a cut
 - Finding an optimal solution requires solutions to multiple subproblems

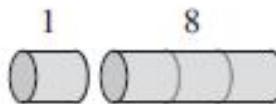
Length I	1	2	3	4	5	6	7	8	9	10
Price P	1	5	8	9	10	17	17	20	24	30

Optimal Rod Cutting Problem

- Consider the **rod of length 4**. the below figure shows all the ways to cut up a rod of 4 inches in length, including the way with no cuts at all.



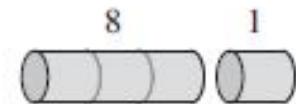
(a)



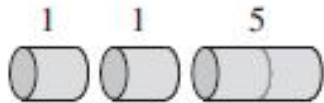
(b)



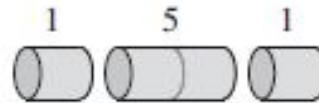
(c)



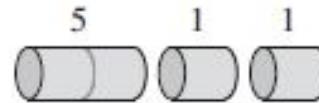
(d)



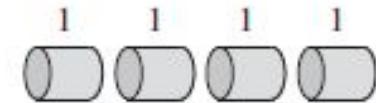
(e)



(f)



(g)



(h)

- We see that cutting a 4-inch rod into two **2-inch pieces** produces revenue $p_2 + p_2 = 5 + 5 = 10$, which is optimal.
- Where we Cut ?**
 - We can cut up a rod of length n in 2^{n-1} different ways, since we have an independent option of cutting, or not cutting, at distance i inches from the left end, for $i=1,2,\dots,n-1$.

Optimal Revenue decompositions

Length I	1	2	3	4	5	6	7	8	9	10
Price P	1	5	8	9	10	17	17	20	24	30

- The above representations can be written as:
 - R1 = result is straight forward
 - R2 = $\max(p_2, p_1+p_1)$
 - R3 = $\max(p_3, p_1+p_2, p_2+p_1, p_1+p_1+p_1)$
 - R4 = $\max(p_4, p_1+p_3, p_2+p_2, p_3+p_1, p_1+p_1+p_2, p_1+p_2+p_1, p_2+p_1+p_1, p_1+p_1+p_1+p_1)$
 - In general, we can frame the values r_n for $n \geq 1$ in terms of optimal revenues from shorter rods:

$$r_n = \max (p_n, r_1 + r_{n-1}, r_2 + r_{n-2}, \dots, r_{n-1} + r_1)$$

- The first argument, p_n , corresponds to making no cuts at all and selling the rod of length n as it is.

Optimal Rod Cutting

- The other $n-1$ arguments to max correspond to the maximum revenue obtained by making an initial cut of the rod into two pieces of size i and $n-i$, for each $i=1,2,\dots,n-1$, and then optimally cutting up those pieces further, obtaining revenues r_i and r_{n-i} from those two pieces.
- Once we make the first cut, we may consider the two pieces as independent instances of the rod-cutting problem.
- The overall optimal solution incorporates optimal solutions to the two related subproblems, maximizing revenue from each of those two pieces.
- We say that the rod-cutting problem exhibits **optimal substructure**: optimal solutions to a problem incorporate optimal solutions to related subproblems, which we may solve independently.

Calculating Maximum Revenue

i	r_i	Maximum of
0	r_0	0
1	r_1	$P_1 + r_0$
2	r_2	$P_1 + r_1, P_2 + r_0$
3	r_3	$P_1 + r_2, P_2 + r_1, P_3 + r_0$
i	r_i

ORCP calculating max revenue

Length I	1	2	3	4	5	6	7	8	9	10
Price P	1	5	8	9	10	17	17	20	24	30

i r_i optimal solution

1 1 1 (no cuts)

2 5 2 (no cuts)

3 8 3 (no cuts)

4 10 2 + 2

5 13 2 + 3

6 17 6 (no cuts)

7 18 1 + 6 or 2 + 2 + 3

8 22 2 + 6

9 25 3 + 6

10 30 10 (no cuts)

Calculating Maximum Revenue

How do we fill in table entry
 r_k :

- For each possible first cut (ie $p_1..p_k$)
- Calculate the sum of the value of that cut (ie p_i) and the best that could be done with the rest of the rod (ie r_{k-i}).
- Choose the largest sum ($p_i + r_{k-i}$).
- Notice that each value of r_i depends only on values higher in the table

i	r_i	Maximum of
0	r_0	0
1	r_1	$P_1 + r_0$
2	r_2	$P_1 + r_1, P_2 + r_0$
3	r_3	$P_1 + r_2, P_2 + r_1, P_3 + r_0$
i	r_i

Example computation

$$\begin{aligned}r_4 &= \max(p_1 + r_3, p_2 + r_2, p_3 + r_1, p_4 + r_0) \\&= \max(1+8, 5+5, 8+1, 9+0) \\&= \max(9, 10, 9, 9) = 10\end{aligned}$$

$$\begin{aligned}r_5 &= \max(p_1 + r_4, p_2 + r_3, p_3 + r_2, p_4 + r_1, p_5 + r_0) \\&= \max(1+10, 5+8, 8+5, 9+1, 10+0) \\&= \max(11, 13, 13, 10, 10) = 13\end{aligned}$$

Optimal Rod Cutting

- If an optimal solution cuts the rod into k pieces, for some $1 \leq k \leq n$, then an optimal decomposition $n = i_1 + i_2 + \dots + i_k$ of the rod into pieces of lengths i_1, i_2, \dots, i_k provides maximum corresponding revenue $r_n = p_{i1} + p_{i2} + \dots + p_{ik}$
- For our sample problem, we can determine the optimal revenue figures r_i , for $i=1,2,\dots,10$, by inspection, with the corresponding optimal decompositions.

$$r_1 = 1 \text{ from solution } 1 = 1 \text{ (no cuts) ,}$$

$$r_2 = 5 \text{ from solution } 2 = 2 \text{ (no cuts) ,}$$

$$r_3 = 8 \text{ from solution } 3 = 3 \text{ (no cuts) ,}$$

$$r_4 = 10 \text{ from solution } 4 = 2 + 2 ,$$

$$r_5 = 13 \text{ from solution } 5 = 2 + 3 ,$$

$$r_6 = 17 \text{ from solution } 6 = 6 \text{ (no cuts) ,}$$

$$r_7 = 18 \text{ from solution } 7 = 1 + 6 \text{ or } 7 = 2 + 2 + 3 ,$$

$$r_8 = 22 \text{ from solution } 8 = 2 + 6 ,$$

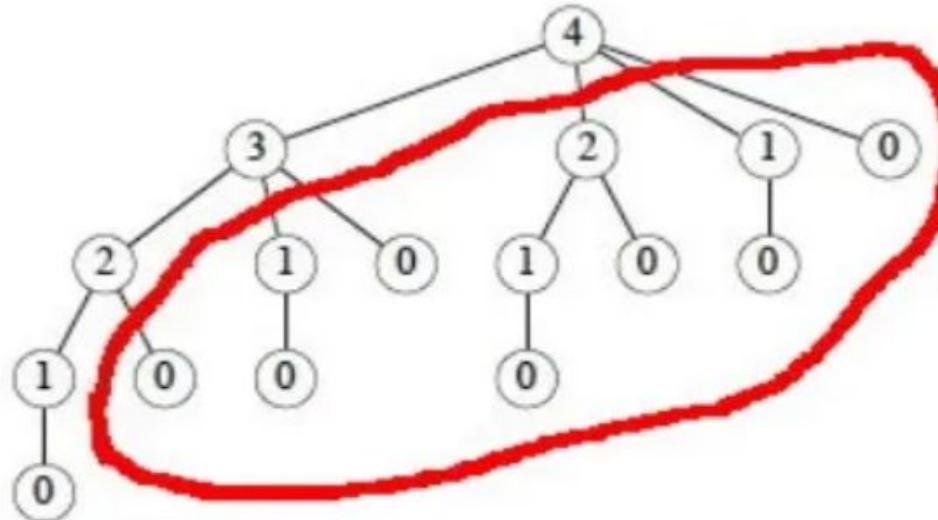
$$r_9 = 25 \text{ from solution } 9 = 3 + 6 ,$$

$$r_{10} = 30 \text{ from solution } 10 = 10 \text{ (no cuts) .}$$

Naïve recursive Rod-Cut

CUT-ROD(p, n)

```
1 if  $n == 0$ 
2     return 0
3  $q = -\infty$ 
4 for  $i = 1$  to  $n$ 
5      $q = \max(q, p[i] + \text{CUT-ROD}(p, n - i))$ 
6 return  $q$ 
```



$$T(n) = 1 + \sum_{j=0}^{n-1} T(j)$$

$$T(n) = 2^n$$

Rod Cutting: Dynamic Programming Solutions

- Problem with recursive solution: Subproblems solved multiple times
- Must figure out a way to solve each subproblem just once
- Two possible solutions: solve a subproblem and remember its solution
 - Top Down: Memoize recursive algorithm
 - Bottom Up: Figure out optimum order to fill the solution array

Dynamic Programming Approaches

- Two Types
 - Top down
 - Bottom up
- Top Down
 - In this approach, we try to solve the bigger problem by recursively finding the solution to smaller sub-problems.
 - Whenever we solve a sub-problem, we cache its result so that we don't end up solving it repeatedly if it's called multiple times. Instead, we can just return the saved result.
 - This technique of storing the results of already solved subproblems is called **Memoization**.

Top Down - Memoize

MEMOIZED-CUT-ROD(p, n)

```
1 let  $r[0..n]$  be a new array
2 for  $i = 0$  to  $n$ 
3      $r[i] = -\infty$ 
4 return MEMOIZED-CUT-ROD-AUX( $p, n, r$ )
```

MEMOIZED-CUT-ROD-AUX(p, n, r)

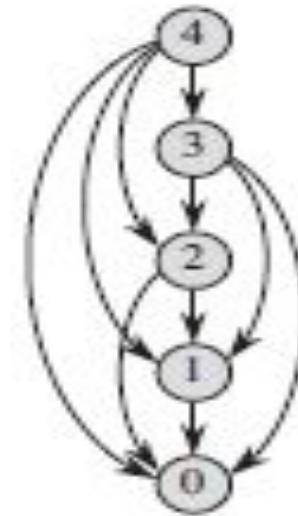
```
1 if  $r[n] \geq 0$ 
2     return  $r[n]$ 
3 if  $n == 0$ 
4      $q = 0$ 
5 else  $q = -\infty$ 
6 for  $i = 1$  to  $n$ 
7      $q = \max(q, p[i] + \text{MEMOIZED-CUT-ROD-AUX}(p, n - i, r))$ 
8  $r[n] = q$ 
9 return  $q$ 
```

Top down

- Here, the main procedure MEMOIZED-CUT-ROD initializes a new auxiliary array $r[0.. n]$ with the value $-\infty$, a convenient choice with which to denote “unknown.”
- It then calls its helper routine, MEMOIZED-CUT-ROD AUX.
- It first checks in line 1 to see whether the desired value is already known and, if it is, then line 2 returns it.
- Otherwise, lines 3–7 compute the desired value q in the usual manner, line 8 saves it in $r[n]$, and line 9 returns it.

Sub problem graphs:

- In dynamic programming problem, the set of sub problems involved depend on one another to get the solution.
- The subproblem graph for the problem embodies exactly this information for e.g.- the subproblem graph for the rod-cutting problem with $n = 4$.
- It is a directed graph, containing one vertex for each distinct subproblem.



Dynamic Programming Approaches

- **Bottom up**

- Tabulation is the opposite of the top-down approach and avoids recursion.
- In this approach, we solve the problem “bottom-up” (i.e. by solving all the related sub-problems first).
- This is typically done by filling up an n-dimensional table.
- Based on the results in the table, the solution to the top/original problem is then computed.

Bottom Up

BOTTOM-UP-CUT-ROD (p, n)

```
1 let  $r[0..n]$  be a new array
2  $r[0] = 0$ 
3 for  $j = 1$  to  $n$ 
4      $q = -\infty$ 
5     for  $i = 1$  to  $j$ 
6          $q = \max(q, p[i] + r[j-i])$ 
7      $r[j] = q$ 
8 return  $r[n]$ 
```

GREEDY METHOD

- Objectives
 - Understand the Design Principle
 - Learn the Control Abstraction / General Method
 - Solve the Examples0 that reveal the intricacies and varieties of the General Method.
 - Design and Analyze the algorithms Associated with Examples.

THE GREEDY METHOD

Most Straight forward Approach

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

- *feasible*,
 - *i.e. satisfying the constraints*
- *locally optimal*
 - *with respect to some decision criteria by not worrying about the consequence of the decision in future*
- *greedy*
 - *(in terms of decision measure), and*
- *irrevocable*

THE GREEDY METHOD

- The greedy method can be applied to a variety of problems. Most of these problems have n inputs and require us to obtain a subset of n inputs that satisfies some **constraints**.
- Any subset (of inputs) that satisfies the constraints is known as **feasible solution**.
- We need to find a feasible solution that maximizes or minimizes a given objective function. A feasible solution that does this is called an **Optimal Solution**.
- There is an obvious way to find a **feasible solution**, but not an **optimal one**.

THE GREEDY METHOD

- A greedy algorithm for an optimization problem **always makes the choice that looks best at the moment** and adds it to the current sub solution.
 - **The hope:** a locally optimal choice will lead to a globally optimal solution
 - For some problems, it works
- Greedy algorithms don't always yield optimal solutions but, when they do, they're usually the simplest and most efficient algorithms available.
- The *greedy approach* is also used in the context of **hard (difficult to solve)** problems in order to generate an **approximate solution**.

Greedy Method

- **Subset Paradigm**
 - At each stage a decision is made to include an input into solution or not depending on the feasibility Constraint.
 - This leads to selection of subset of inputs into solution set.
 - Fractional Knapsack, Minimum Cost Spanning Tree, Job Sequencing with deadlines are examples of Subset Paradigm.
- **Ordering Paradigm**
 - At each stage decision are made by considering the inputs in some order.
 - Single Source Shortest Path problem.

THE GREEDY METHOD (Contd..)

Procedure Greedy (A,n)

//A (1:n) contains n inputs //

 solution $\leftarrow \emptyset$

 for $i \leftarrow 1$ to n do

$x \leftarrow \text{SELECT } (A)$

 if feasible (solution, x) then

 solution $\leftarrow \text{union } (\text{solution}, x)$

 end if

 repeat

 return (solution)

END GREEDY

Fractional Knapsack Problem

- In **fractional knapsack problem**, we are given a set S of n items, such that each item i has a **profit** p_i and a **weight** w_i , and we wish to find the **maximum-profit** subset that *doesn't exceed* a given capacity of Bag M .
- We are also allowed to take *arbitrary fractions* of each item.
- In other words i.e., we can take an amount x_i of each item i such that

$$\text{maximize} \sum_{1 \leq i \leq n} p_i x_i \quad (4.1)$$

$$\text{subject to} \sum_{1 \leq i \leq n} w_i x_i \leq m \quad (4.2)$$

$$\text{and } 0 \leq x_i \leq 1, \quad 1 \leq i \leq n \quad (4.3)$$

Greedy Knapsack

- Let $n = 3$, $m= 20$, $(p_1,p_2,p_3)=(25,24,15)$ and $(w_1,w_2,w_3)= (18,15,10)$
- Several Ways of Selection of items into Knapsack to obtain Feasible solutions
 - Select the items randomly
 - Select the items in ascending order of weights (Least Weight)
 - Select the items in descending order of profits (Largest Profit)
 - Select the items in descending order of P/W (Largest Profit/Weight ratio)

Fractional Knapsack – Several ways

Let $n = 3, m = 20$,

$(p_1, p_2, p_3) = (25, 24, 15)$ and

$(w_1, w_2, w_3) = (18, 15, 10)$.

(X_1, X_2, X_3)	$\sum W_i x_i$	$\sum P_i x_i$
$(1/2, 1/3, 1/4)$	16.5	24.25
$(1, 2/15, 0)$	20	28.2
$(0, 2/3, 1)$	20	31
$(0, 1, 1/2)$	20	31.5

- Lemma
 - In case the sum of all the weights is $\leq m$ then all $x_i = 1, 1 \leq i \leq n$ is an optimal solution
 - All optimal solutions will fill the knapsack exactly.

Fractional Knapsack – Selection Strategy

- First, find the profit and weight ratios of the objects and sort in the descending order of the ratios.
- Select an object with highest p/w ratio and check whether its weight is lesser than the capacity of the bag.
 - If so place 1 unit of the object and decrement the capacity of the bag by the weight of the object you have placed.
- Repeat the above steps until the capacity of the bag becomes less than the weight of the object selected . At this point place a fraction of the object and come out of the loop.

Fractional Knapsack

- GreedyKnapSack(m,n)

{

for i = 1 to n do x[i]=0;

U=m;

for i = 1 to n do

{

if (w[i] > U) then break;

else

{x[i] = 1;

U = U-w[i];}

}

if (i <=n) then x[i] = U/w[i];

The Time complexity of Fractional Knapsack is O(n)

Tracing Fractional Knapsack

$n=3, m=20, (p_1, p_2, p_3)=(25, 24, 15),$
 $(w_1, w_2, w_3)=(18, 15, 10)$

The decreasing order of p_i/w_i is : $(24/15, 15/10, 25/18) = (1.6, 1.5, 1.39)$

Tracing

Initially set $x[i]=0$ ie $x[1]=0.0, x[2]=0.0, x[3]=0.0$

Given $m=20, U=m$ therefore $U=20$

$i=1$

if ($w[1] > 20$)

$15 > 20 \dots \text{false, then } x[1]=1.0$

$U=20-w[1] = 20-15 = 5$

Therefore remaining capacity is $U=5 \dots \text{goes back to for loop with } i=2$

if ($w[2] > 5$)

$10 > 5 \dots \text{true, break}$

if ($i \leq n$)

$(2 \leq 3) \dots \text{true}$

$x[2]=U/w[2]=5/w[2] \text{ ie } x[2]=5/10=1/2$

Therefore, the optimal solution is : $(1, 1/2, 0)$ and the profit is : $24+15 * 1/2 = 31.5$

Exercise

- $n=7, m=15,$
 $(p_1, p_2, p_3, p_4, p_5, p_6, p_7) = (10, 5, 15, 7, 6, 18, 3)$
 $(w_1, w_2, w_3, w_4, w_5, w_6, w_7) = (2, 3, 5, 7, 1, 4, 1)$

Job Sequencing with Dead Lines

The problem is stated as below.

- Problem: n jobs, $S=\{1, 2, \dots, n\}$, each job i has a deadline $d_i \geq 0$ and a profit $p_i \geq 0$. We need one unit of time to process each job and we can do at most one job each time. The profit p_i is earned if job i is completed by its deadline.
- We have to find **processing sequence of jobs** so that such a sequence completes the jobs before deadline and **earns maximum profit**.
- A feasible solution is a subset of jobs J such that each job is completed by its deadline.
- An optimal solution is a feasible solution with maximum profit value.

Job Sequencing with Deadlines

- Let (2,3,1,3) are the given deadlines of 4 jobs
 - Each job can be completed in 1 unit of time.
 - Job 1 should be completed before the deadline of 2. That means it can be delayed by 1 unit of time. Implies we can complete any **one job** before job 1.
 - Job 4 should be completed before the deadline of 3. That means it can be delayed by 2 units of time. Implies we can complete any **two jobs** before job 4.
- The objective is to select jobs that maximize profit.
- A feasible solution is a subset J of jobs such that each job in this subset can be completed by its deadline.
- Since the problem involves the identification of a subset, it fits the subset paradigm.

Job Sequencing with Dead Lines

Example : Let $n = 4$, $(p_1, p_2, p_3, p_4) = (100, 10, 15, 27)$, $(d_1, d_2, d_3, d_4) = (2, 1, 2, 1)$

Sr.No.	Feasible Solution	Processing Sequence	Profit value
(i)	(1,2)	(2,1)	110
(ii)	(1,3)	(1,3) or (3,1)	115
(iii)	<u>(1,4)</u>	<u>(4,1)</u>	<u>127</u> ↑ is the optimal one
(iv)	(2,3)	(2,3)	25
(v)	(3,4)	(4,3)	42
(vi)	(1)	(1)	100
(vii)	(2)	(2)	10
(viii)	(3)	(3)	15
(ix)	(4)	(4)	27

Algorithm:

Step 1: Sort p_i into nonincreasing order. After sorting $p_1 \geq p_2 \geq p_3 \geq \dots \geq p_i$.

Step 2: Add the next job i to the solution set if i can be completed by its deadline. Assign i to time slot $[r-1, r]$, where r is the largest integer such that $1 \leq r \leq d_i$ and $[r-1, r]$ is free.

Step 3: Stop if all jobs are examined. Otherwise, go to step 2.

Time complexity: $O(n^2)$

GREEDY ALGORITHM TO OBTAIN AN OPTIMAL SOLUTION

- Consider the jobs in the non increasing order of profits subject to the constraint that the resulting job sequence J is a feasible solution.
- In the example considered before, the non-increasing profit vector is

Job _i	P _i	D _i	slot
1	100	3	assign [2-3]
2	27	1	assign[0-1] slot
3	15	1	reject
4	10	2	assign[1-2] slot

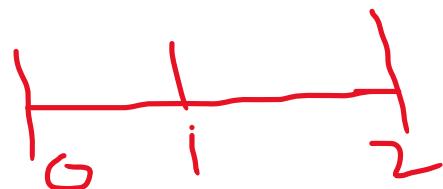
Solution {1,2,4}
Profit is 137

JSD example

Let Jobs with profits $(p_1, p_2, p_3, p_4) = (100, 10, 15, 27)$ and deadlines $(d_1, d_2, d_3, d_4) = (2, 1, 2, 1)$

Rearrange as $(p_1, p_4, p_3, p_2) = (100, 27, 15, 10)$

$(d_1, d_4, d_3, d_2) = (2, 1, 2, 1)$



$\{J\}$	Assigned Slot	Profit Earned	Job Considered	Action
Empty (initially)	None	0	1	Assign to [1,2]
{1}	[1-2]	100	4	Assign to [0,1]
{1, 4}	[0-1][1-2]	$100 + 27 = 127$	3	Reject, no slot to complete job 3.
{1, 4}	[0-1][1-2]	127	2	Reject
{1, 4}	[0-1][1-2]	127	--	--

Job Sequencing with Dead Lines Algorithm

```
1  Algorithm JS( $d, j, n$ )
2    //  $d[i] \geq 1$ ,  $1 \leq i \leq n$  are the deadlines,  $n \geq 1$ . The jobs
3    // are ordered such that  $p[1] \geq p[2] \geq \dots \geq p[n]$ .  $J[i]$ 
4    // is the  $i$ th job in the optimal solution,  $1 \leq i \leq k$ .
5    // Also, at termination  $d[J[i]] \leq d[J[i + 1]]$ ,  $1 \leq i < k$ .
6  {
7     $d[0] := J[0] := 0$ ; // Initialize.
8     $J[1] := 1$ ; // Include job 1.
9     $k := 1$ ;
10   for  $i := 2$  to  $n$  do
11   {
12     // Consider jobs in nonincreasing order of  $p[i]$ . Find
13     // position for  $i$  and check feasibility of insertion.
14      $r := k$ ;
15     while  $((d[J[r]] > d[i])$  and  $(d[J[r]] \neq r))$  do  $r := r - 1$ ;
16     if  $((d[J[r]] \leq d[i])$  and  $(d[i] > r))$  then
17     {
18       // Insert  $i$  into  $J[ ]$ .
19       for  $q := k$  to  $(r + 1)$  step  $-1$  do  $J[q + 1] := J[q]$ ;
20        $J[r + 1] := i$ ;  $k := k + 1$ ;
21     }
22   }
23   return  $k$ ;
24 }
```

Tracking

Profits is decreasing order: (20,15,10,5,1) deadlines= (2,2,1,3,3)

d[0]=0; J[0]=0; J[1]=1;

K=1

For loop i=2 {

r=1

while((d[J[1]]>d[2] && (d[J[1]]≠ 1)) (2 > 2 && 2 ≠ 1)---false

if ((d[J[1]] ≤ d[i]) && (d[i]>r)) (2 ≤ 2 && 2 >1)----true

{

q=1 to 2 step -1

q=2 ==> J[2+1]=J[q] ie j[3]=j[2] , iterate back in for

q=1 J[2]=J[1]

J[r+1]=i J[2]=2 k=k+1=2

}

goes to for loop. i=3

{

r=2;

while((d[J[2]]>d[3] &&(d[J[2]] ≠2)) (2>1 && 2 ≠2)-----false

Tracing

```
if ((d[j[r]] ≤ d[i]) && (d[i]>r))  
(d[j[2]] ≤ d[3]) && d[3] >2 2 ≤1 && 1>2----false
```

goes back to for i=4

```
{ r=2  
while ((d[j[2]]>d[4]) & (d[j[2]]≠2)) (2 > 3 && 2 ≠2)----false  
if ((d[j[2]] ≤ d[4]) && (d[4] >2))  
(2 ≤ 3 ) && 3 >2-----true {  
q:=2 to 3 step -1 q:=3, J[4]=J[3]  
q=2, J[3]=J[2] J[3]=4 k=k+1=3 }
```

Goes back to for l =5

r=3

```
while ((d[J[3]] >d[5]) & d[J[3]]≠ 3)) 1 >3-----false  
If((d[J[3]] ≤ d[5]) & (d[5] >3))  
(1 ≤ 3 && 3 >3)----false comes out of algorithm
```

therefore the optimal solution is J {1,2,4} with profit: 40

Time Complexity Analysis

- The while loop at line 15 checks for the feasibility of job Inclusion
 - Time required for this is atmost k time say $\Theta(k)$
- Lines 19 to 20 to insert the job, if feasible
 - Time required for this is $\Theta(k - r)$
- Total time for each iteration of for loop at Line 10 requires $\Theta(k)$ time.
- If s is the number of jobs included in $J[]$, then the total time needed is $\Theta(sn)$. Since $s \leq n$, the worst case time is $O(n^2)$.

Exercise

	J1	J2	J3	J4	J5	J6	J7	J8	J9
Profit	15	20	30	18	18	10	23	16	25
Dead lines	7	2	5	3	4	5	2	7	3

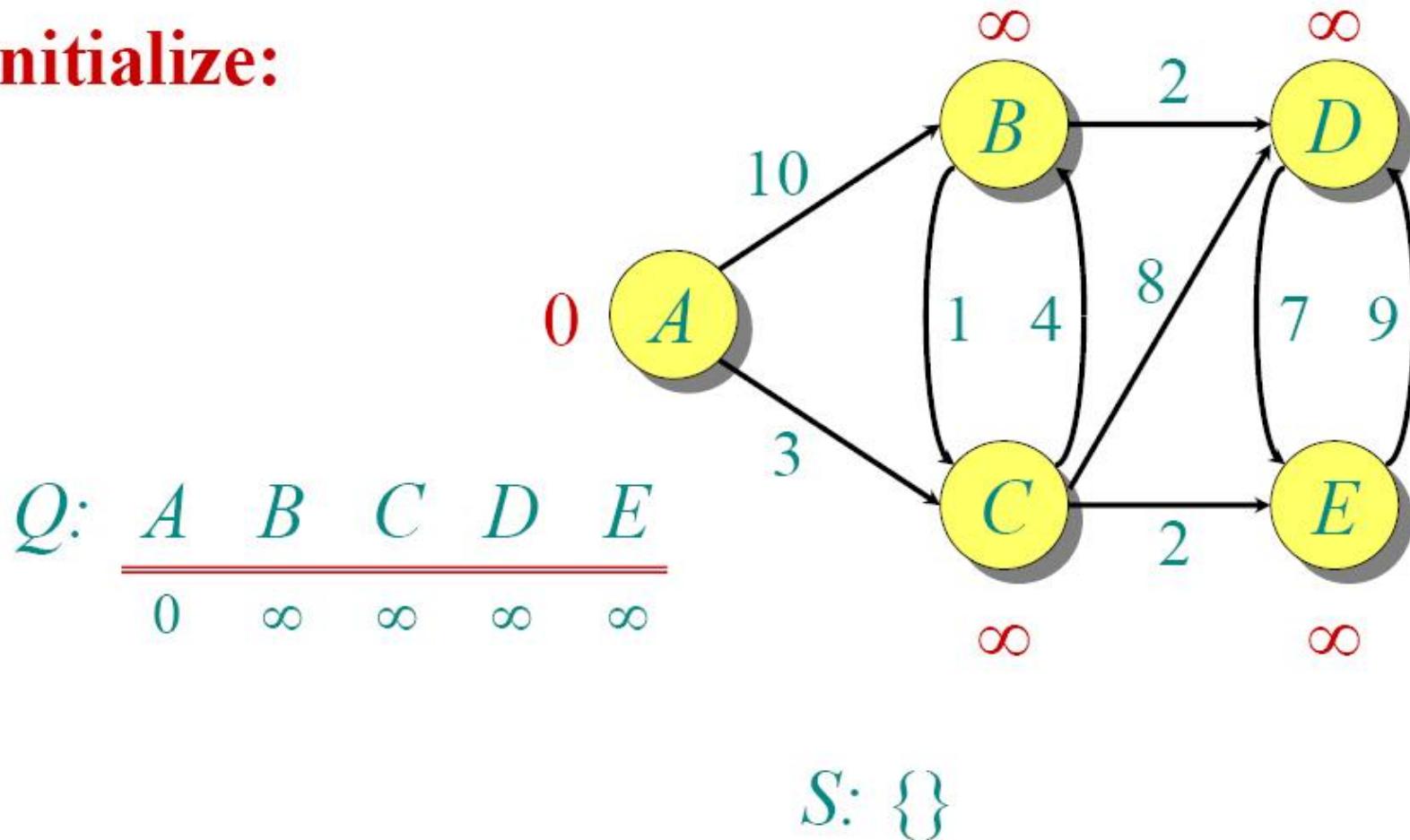
Find the Solution to the following Job Sequencing with deadlines problem

Dijkstras Single Source Shortest Path

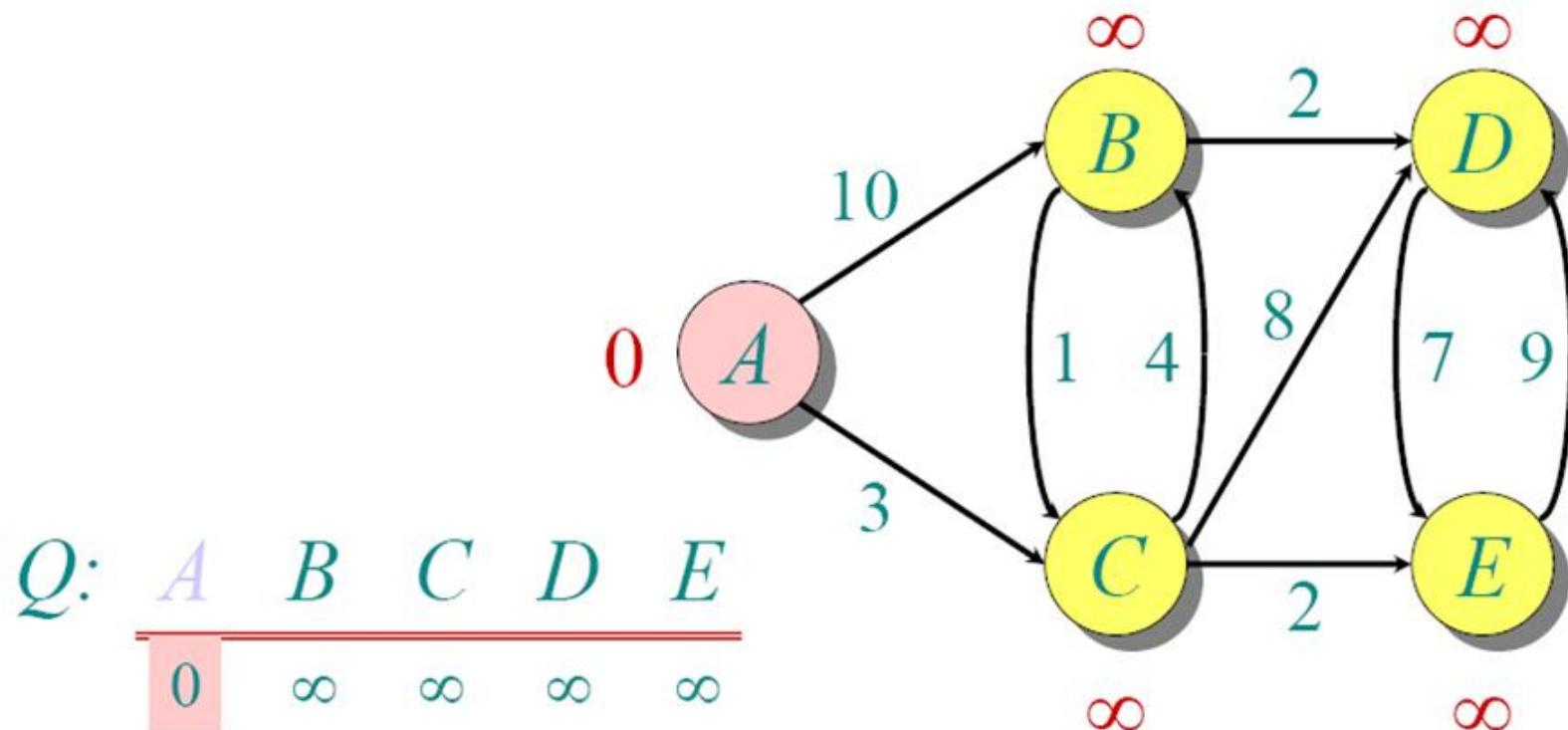
- Given a directed Graph $G = (V,E)$, V is set of vertices, E set of edges, with edges assigned cost and the source vertex V_0 . The problem is to determine the shortest paths from V_0 to all the remaining vertices of G .
- It is called Dijkstras SSS Path problem,
- It can also be used for finding costs of shortest paths from a single vertex to a single destination vertex by stopping the algorithm once the shortest path to the destination vertex has been determined.

Dijkstra Animated Example

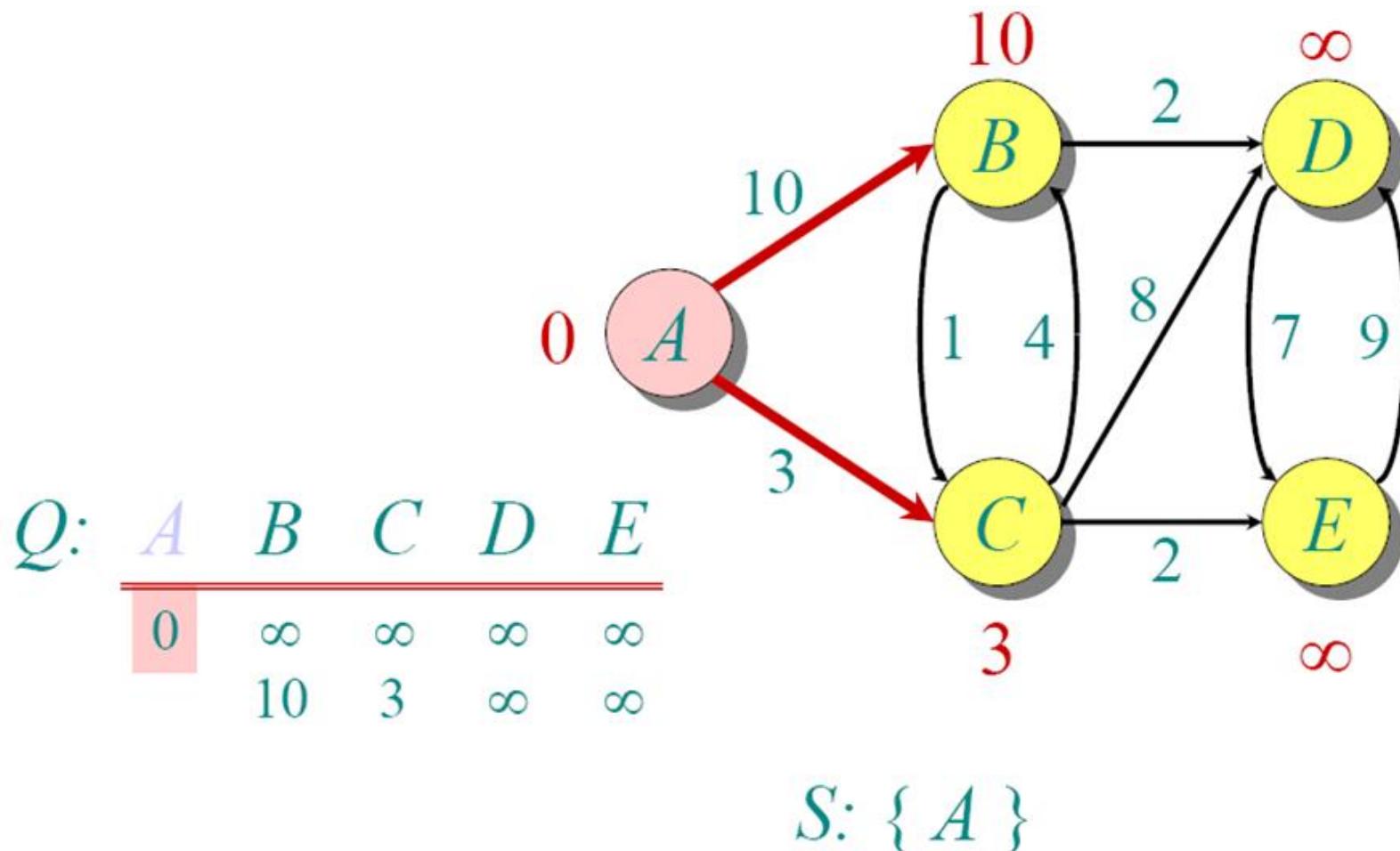
Initialize:



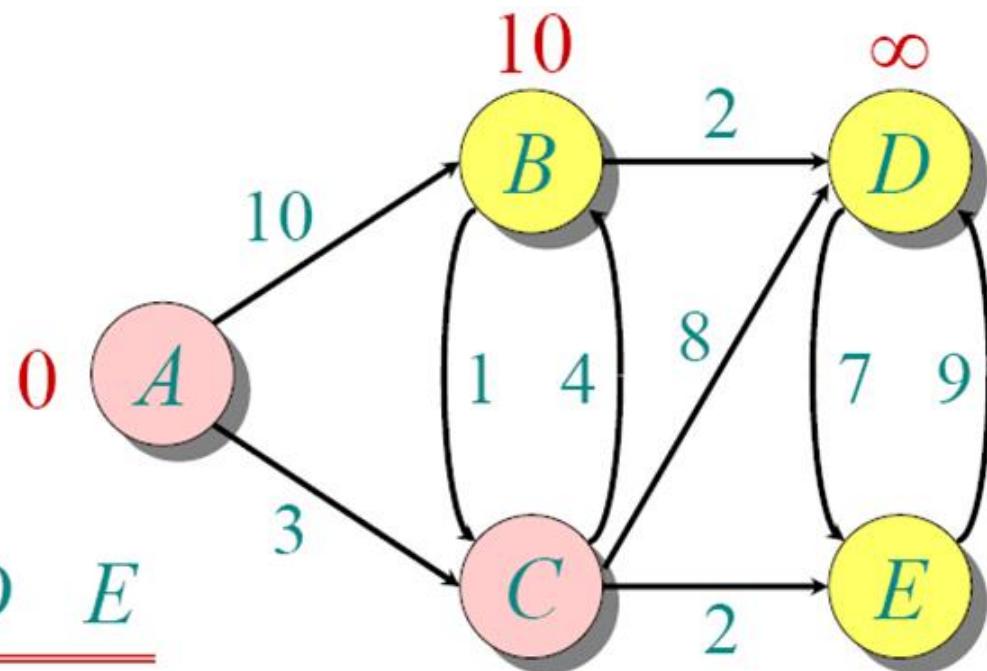
Dijkstra Animated Example



Dijkstra Animated Example



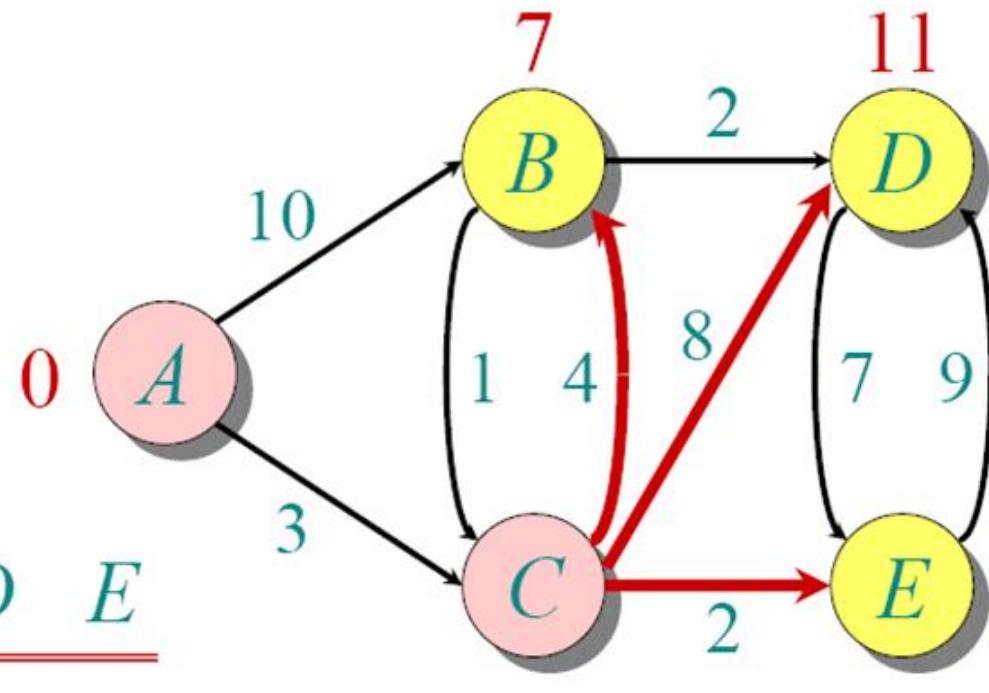
Dijkstra Animated Example



$Q:$	A	B	C	D	E
	0	∞	∞	∞	∞
	10	3	∞	∞	∞

$S: \{ A, C \}$

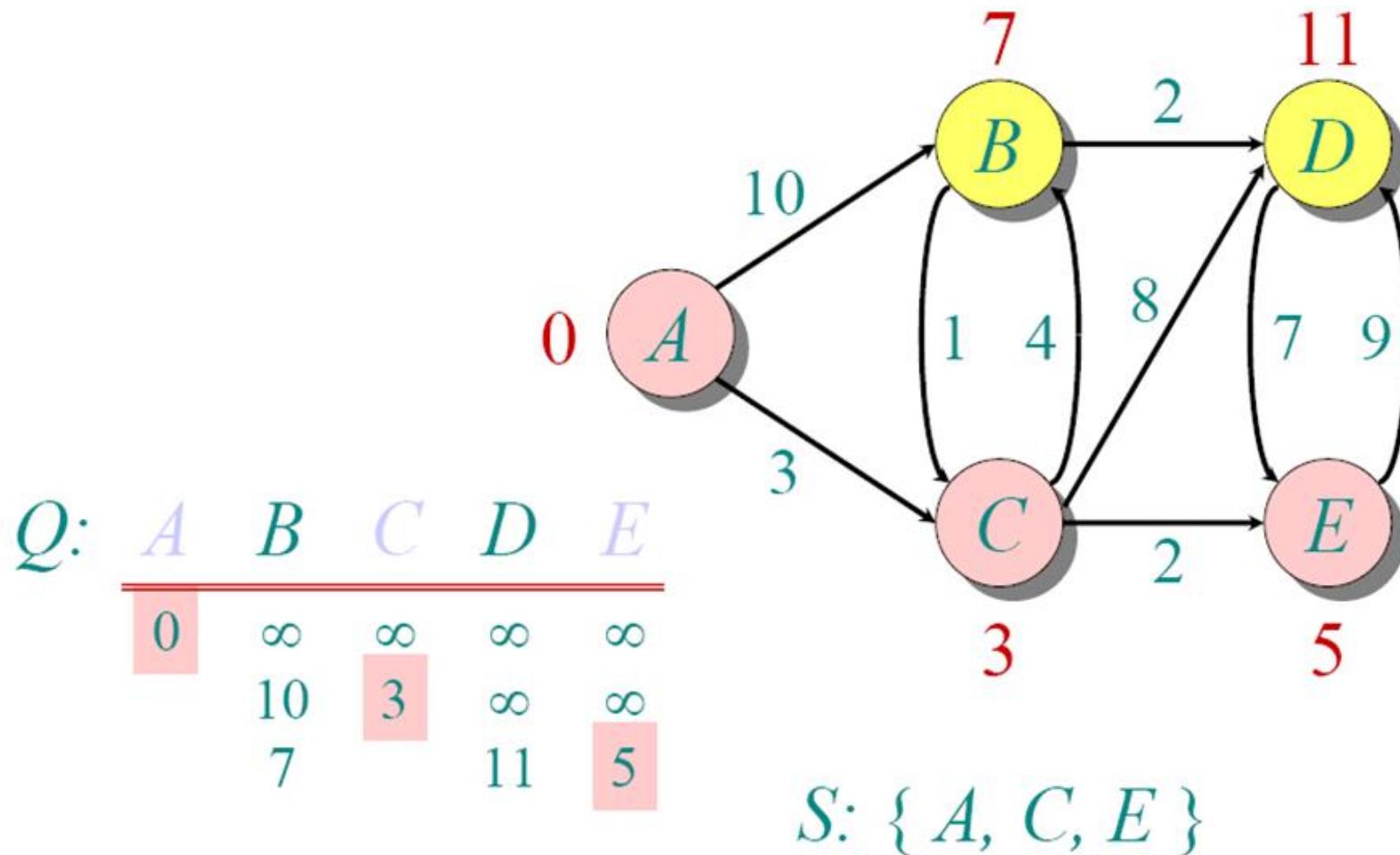
Dijkstra Animated Example



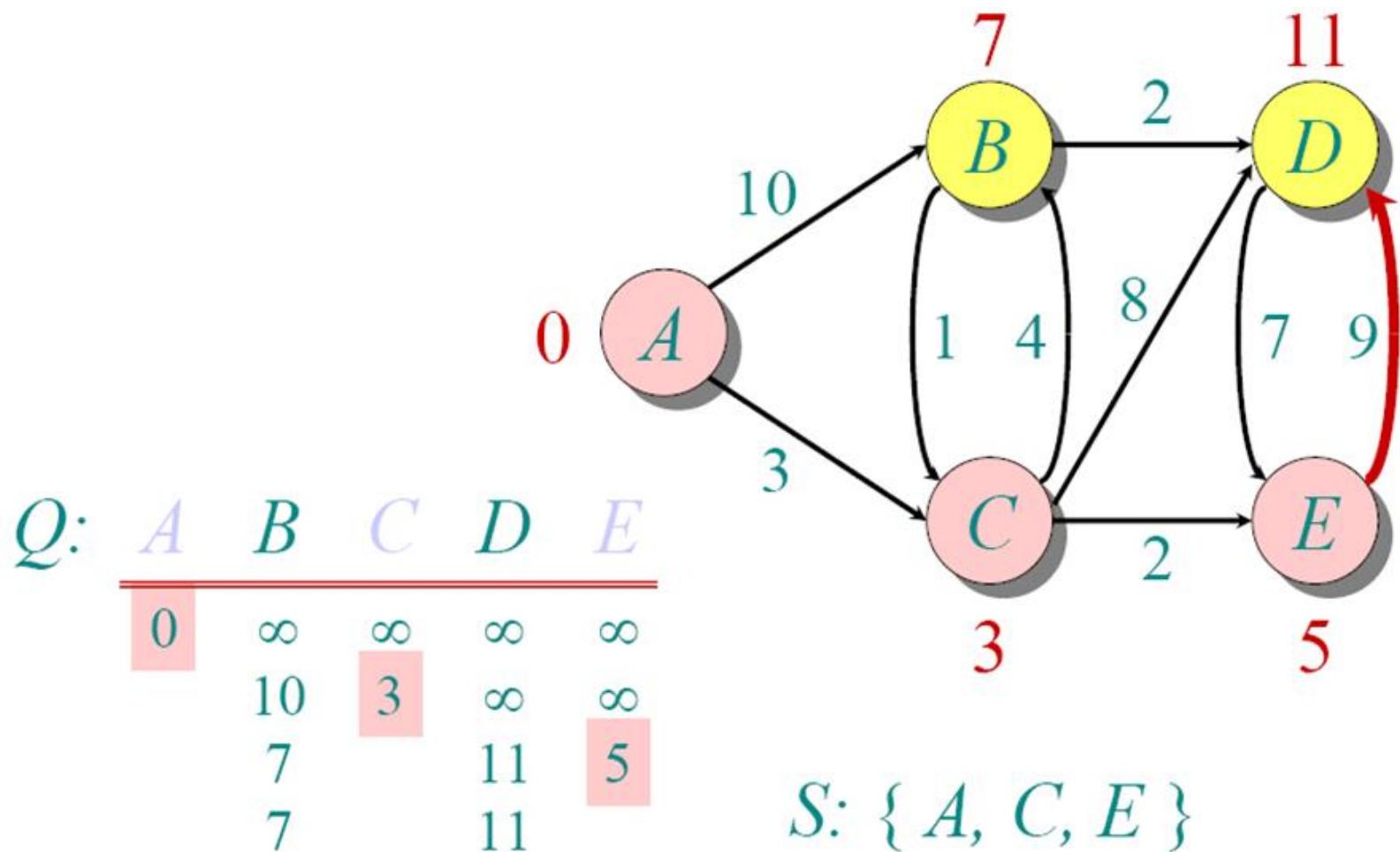
$Q:$	A	B	C	D	E
0	∞	∞	∞	∞	∞
10	3	∞	∞	∞	
7		11	5		

$S: \{ A, C \}$

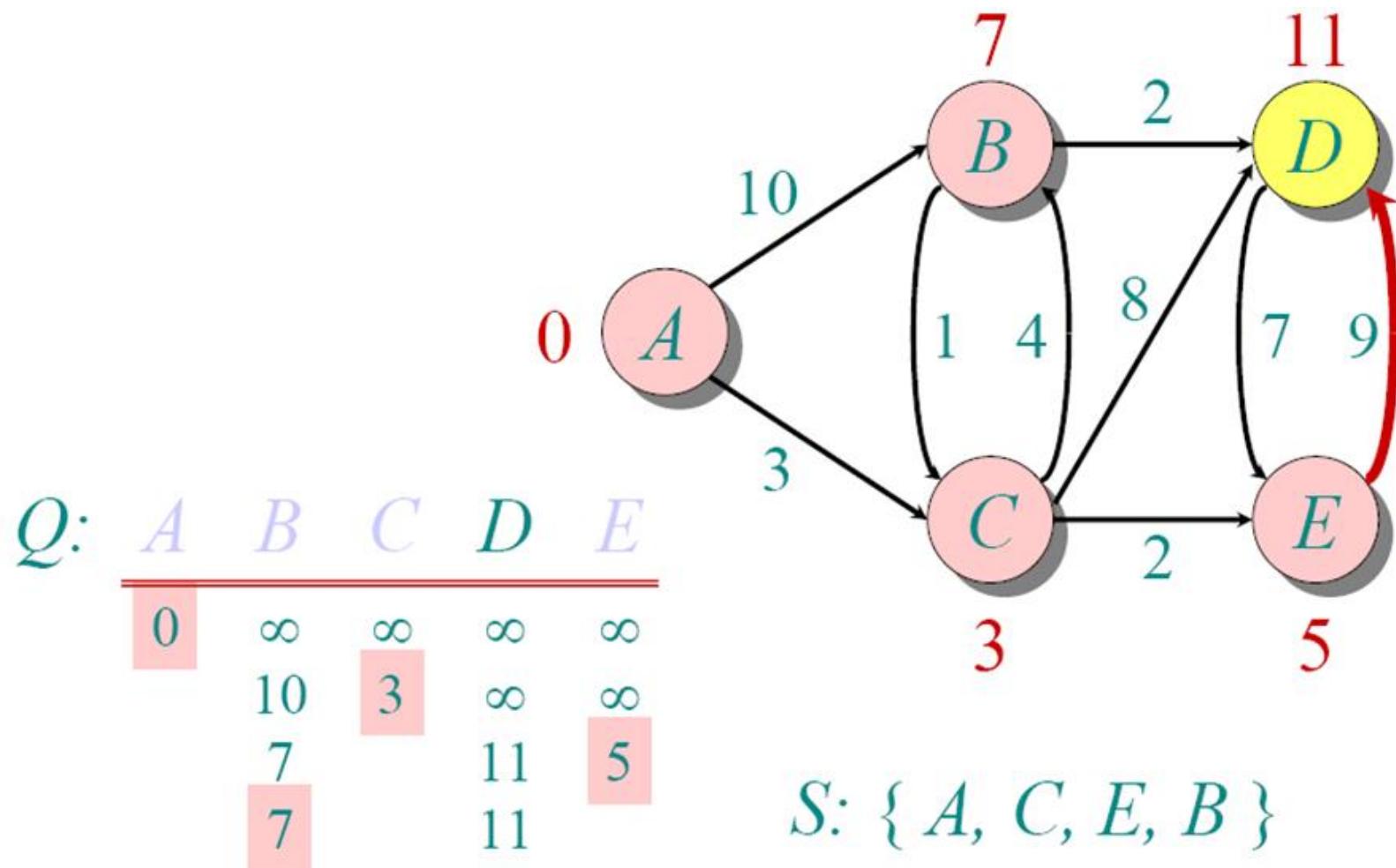
Dijkstra Animated Example



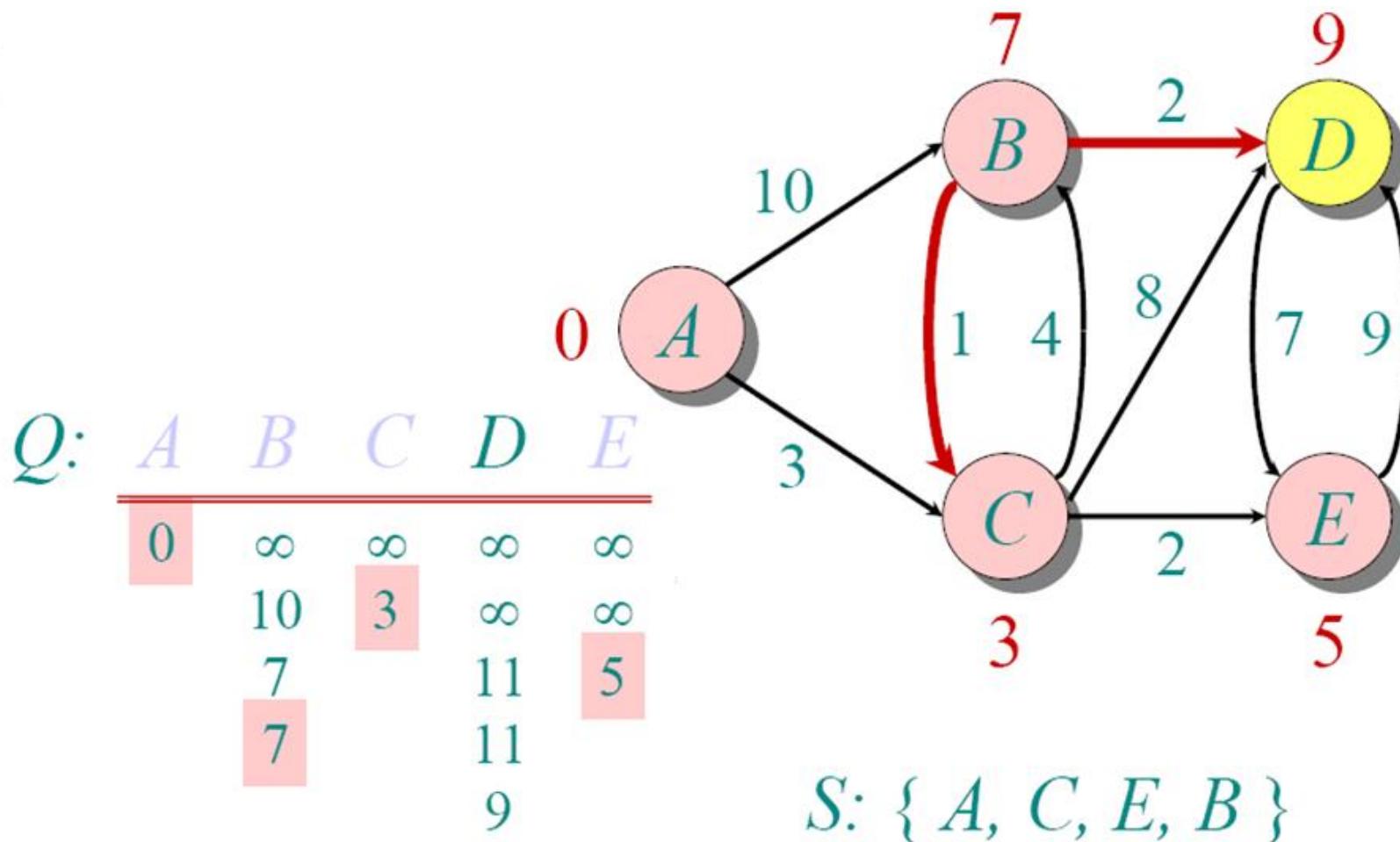
Dijkstra Animated Example



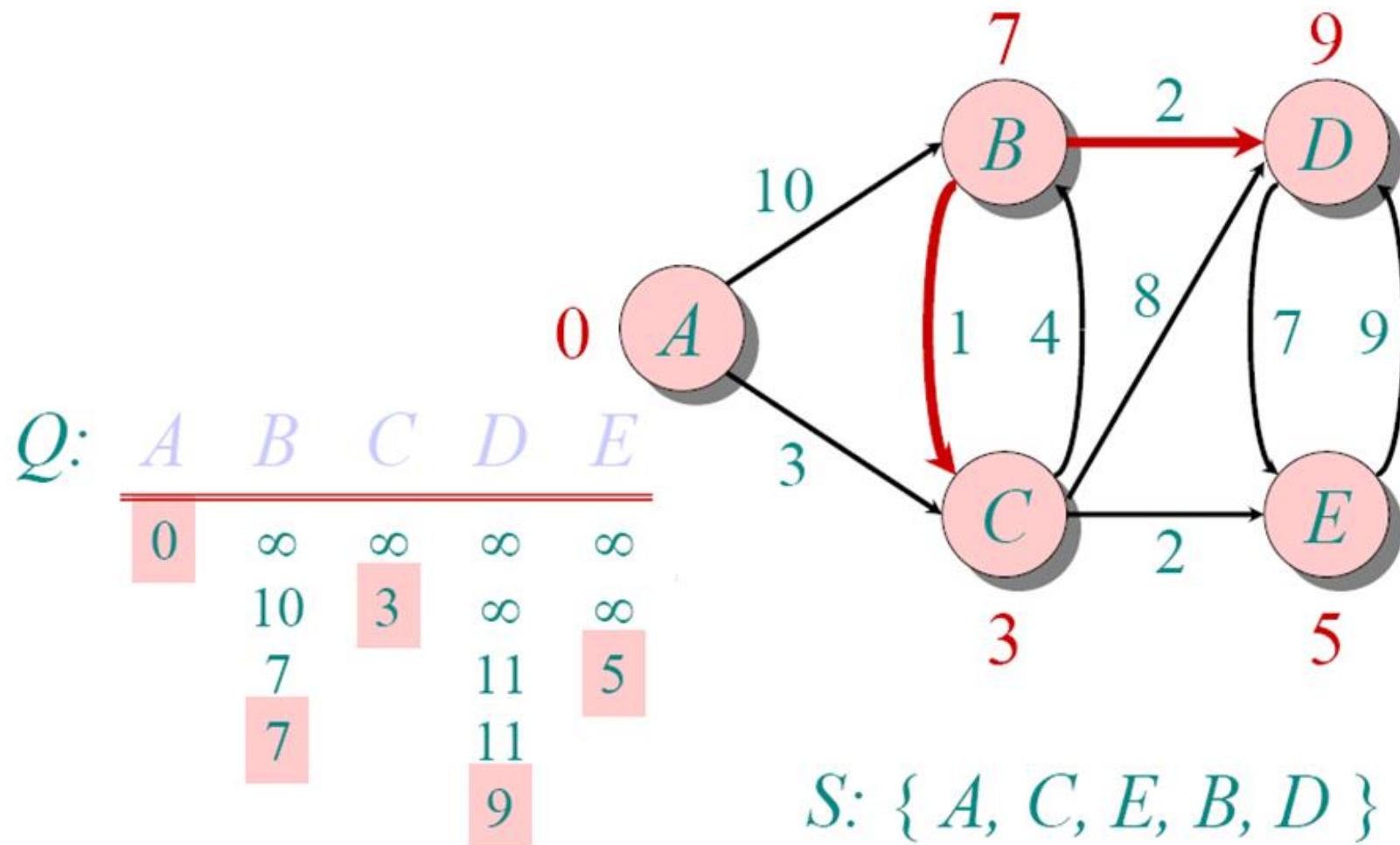
Dijkstra Animated Example



Dijkstra Animated Example



Dijkstra Animated Example



Single-Source Shortest Paths: Dijkstra's Algorithm

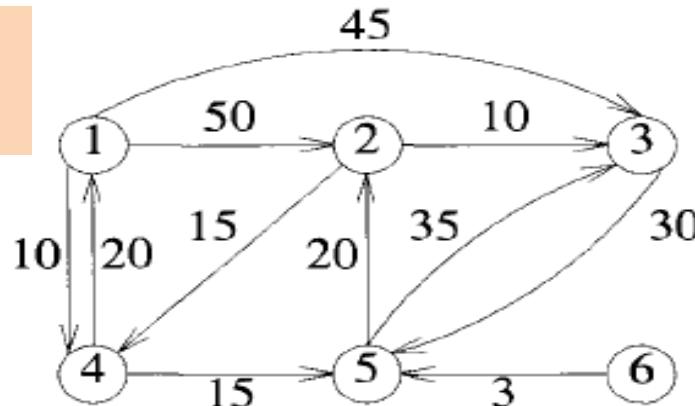
```
1  Algorithm ShortestPaths( $v, cost, dist, n$ )
2    //  $dist[j]$ ,  $1 \leq j \leq n$ , is set to the length of the shortest
3    // path from vertex  $v$  to vertex  $j$  in a digraph  $G$  with  $n$ 
4    // vertices.  $dist[v]$  is set to zero.  $G$  is represented by its
5    // cost adjacency matrix  $cost[1 : n, 1 : n]$ .
6  {
7    for  $i := 1$  to  $n$  do
8      { // Initialize  $S$ .
9         $S[i] := \text{false}$ ;  $dist[i] := cost[v, i]$ ;
10       }
11       $S[v] := \text{true}$ ;  $dist[v] := 0.0$ ; // Put  $v$  in  $S$ .
12      for  $num := 2$  to  $n - 1$  do
13      {
14        // Determine  $n - 1$  paths from  $v$ .
15        Choose  $u$  from among those vertices not
16        in  $S$  such that  $dist[u]$  is minimum;
17         $S[u] := \text{true}$ ; // Put  $u$  in  $S$ .
18        for (each  $w$  adjacent to  $u$  with  $S[w] = \text{false}$ ) do
19          // Update distances.
20          if ( $dist[w] > dist[u] + cost[u, w]$ ) then
21             $dist[w] := dist[u] + cost[u, w]$ ;
22        }
23      }
```

Time Complexity

- The time require to initialize the distance and S arrays is $O(n)$
- The outer for loop of line 12 runs approximately $O(n)$ times
- Inside the loop
 - Each execution of this loop requires to find the minimum cost vertex not yet included in S, this takes distance array to be examined, which takes $O(n)$ time
 - Updating the cost of all adjacent vertices to the new minimum vertex , which again takes $O(n)$ time
- Overall Time : $O(n^2)$

Try the following for Single Source Shortest path Problem

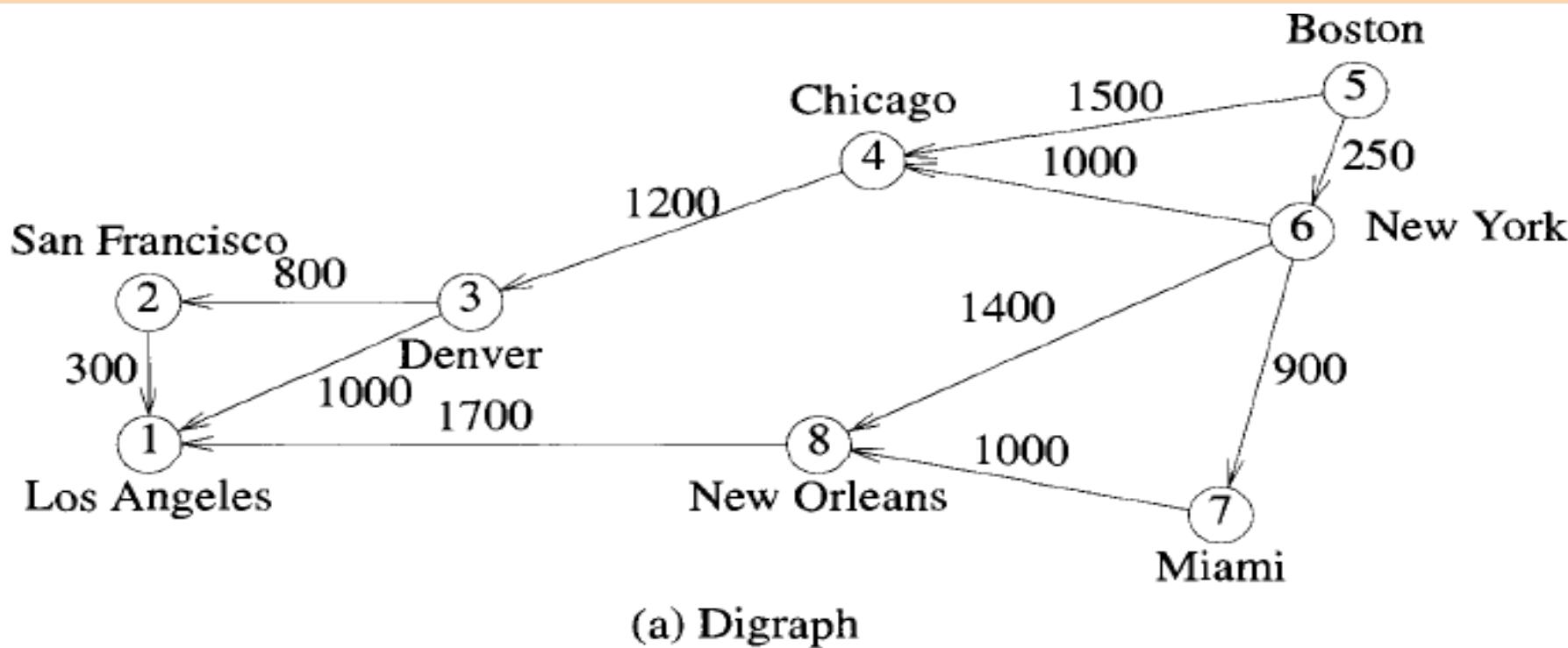
1. Source Vertex is **1**.



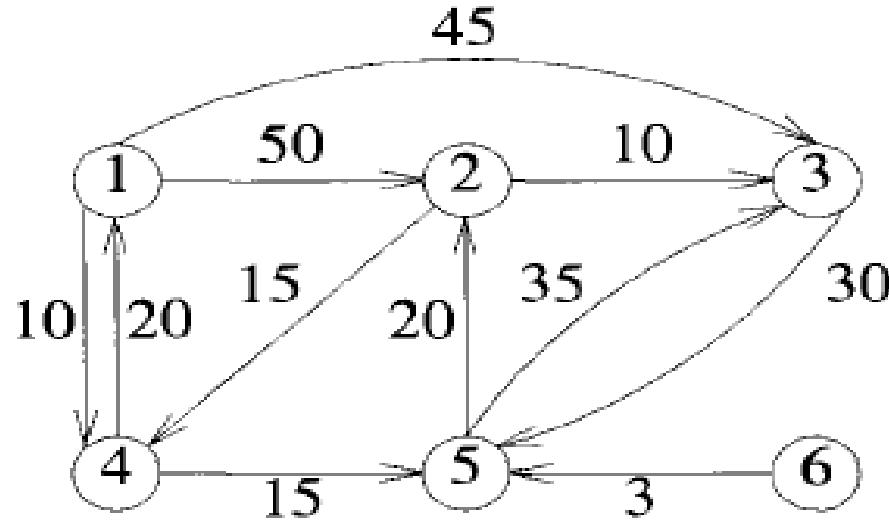
<i>Path</i>	<i>Length</i>
1) 1, 4	10
2) 1, 4, 5	25
3) 1, 4, 5, 2	45
4) 1, 3	45

(b) Shortest paths from 1

2. Source Vertex is **5 Boston**.



Tracing



(a) Graph

	1	2	3	4	5	6
1	0	50	45	10	∞	∞
2	∞	0	10	15	∞	∞
3	∞	∞	0	∞	30	∞
4	20	∞	∞	0	15	∞
5	∞	20	35	∞	0	∞
6	∞	∞	∞	∞	3	0

$d[j]$ denotes the length of shortest path from vertex v to vertex j in a digraph with n vertices

initially $dist[i]$ is set to $cost[v,i]$

let vertex $v = 1$

$d[i] = cost[1,i]$ where $1 \leq i \leq n$, n is the vertices in G

0	50	45	10	∞	∞
---	----	----	----	----------	----------

$S = \{1\} \implies$ vertex 1 is included in solution set.

Tracing

0	50	45	10	25	∞
---	----	----	----	----	----------

num = 2

for loop

consider the vertex ' u' with minimum distance from d[], which is not in S ==>

u = 4

S= {1,4}

consider adjacent vertices of 4

adj to 4 not in S are vertex 5 , therefore vertex 5 is considered.

is $d[5] > d[4]+cost[4,5]$ ==> $\infty > 10 + 15 = 25 ==> d[5] > 25$ thus $d[5]$ is assigned 25

num =3

consider the vertex ' u' with minimum distance from d[], which is not in S ==>

u = 5

S= {1,4,5}

consider adjacent vertices of 5

adj to 5 not in S are vertex 2 and vertex 3

is $d[2]> d[5]+cost[5,2]$ ===== $50 > 25 + 20 = 45 ==> d[2] > 45$ thus $d[2] = 45$
assigned 45

is $d[3] > d[3]+cost[5,3]$ ===== $45 > 25 + 35 = 60 ==> d[3] ! > 45$ thus $d[3]$ is
as it is - no change

0	45	45	10	25	.
---	----	----	----	----	---

Tracing

num = 4

consider the vertex ' u' with minimum distance from dist[], which is not in S ==>

u = 2

S= {1,4,5,2}

0	45	45	10	25	∞
---	----	----	----	----	----------

consider adjacent vertices of 2

adj to 2 not in S are vertex 3

is $d[3] > d[2] + \text{cost}[2,3]$ ====== $45 > 45 + 10 = 55$ ==> $d[3] ! > 55$ thus $d[3]$ is as it is - no change

num=5 u = 3, S= { 1, 4,5,2, 3}

0	45	45	10	25	∞
---	----	----	----	----	----------

paths

$$\{1,4\} = 10$$

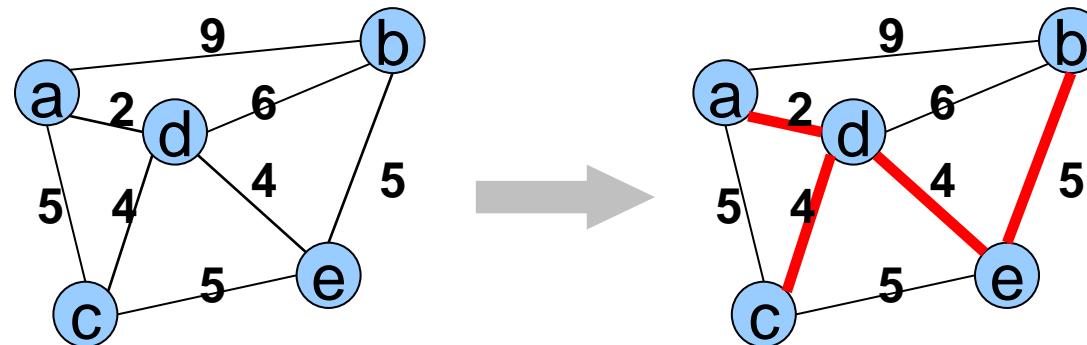
$$\{1,4,5\} = 25$$

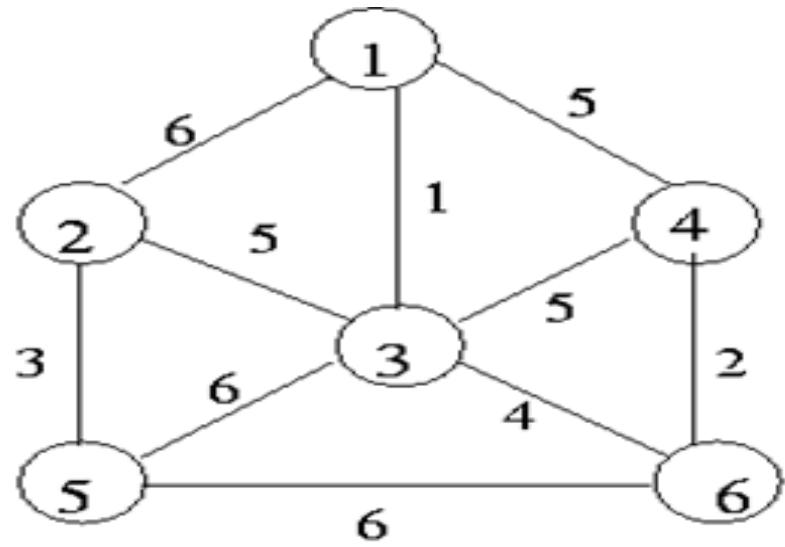
$$\{1,4,5,2\} = 45$$

$$\{1,3\} = 45$$

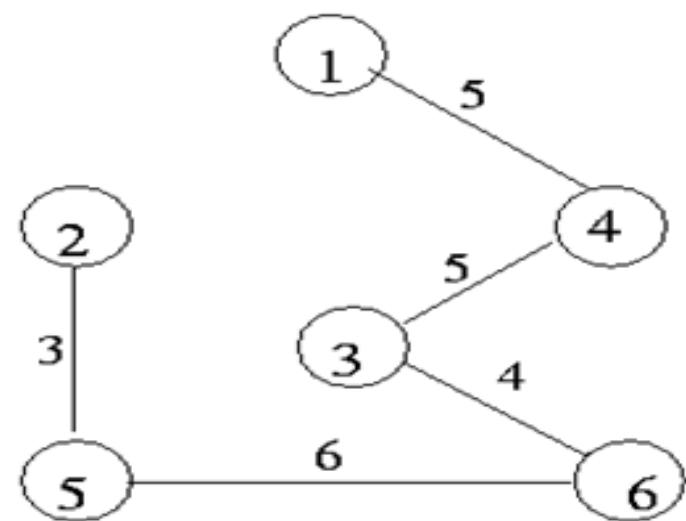
Spanning Tree

- **Spanning Trees:** Let $G = (V, E)$ be an undirected graph, consisting set of Vertices, V and set of Edges E . A sub graph T is a **Spanning Tree** of G if it is a tree and contains every vertex of G .
- A **Minimum Spanning Tree in an undirected connected** weighted graph is a spanning tree of minimum weight (among all spanning trees).
- In a **complete graph** on n vertices we can construct **total $n^{(n-2)}$** different spanning trees.
- **Not necessarily be unique :**However, if the weights of all the edges are pair wise distinct, it is indeed unique (we won't prove this now).

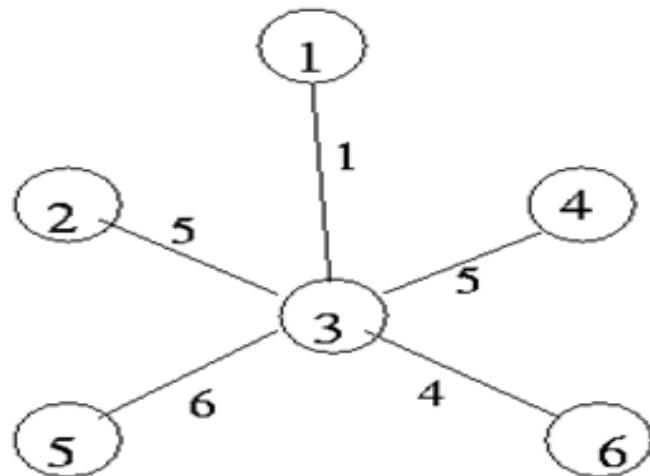




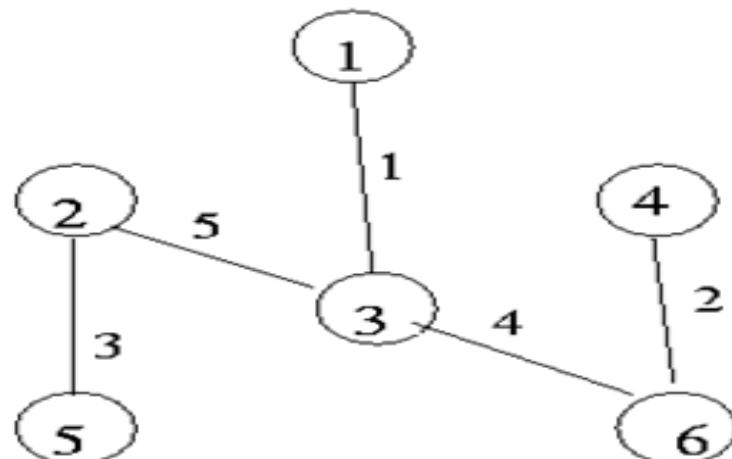
A connected graph.



A spanning tree with cost = 23



Another spanning tree with cost 21



MST, cost = 15

Applications of Spanning Tree

Used to obtain independent set of circuit equations for an electric network.

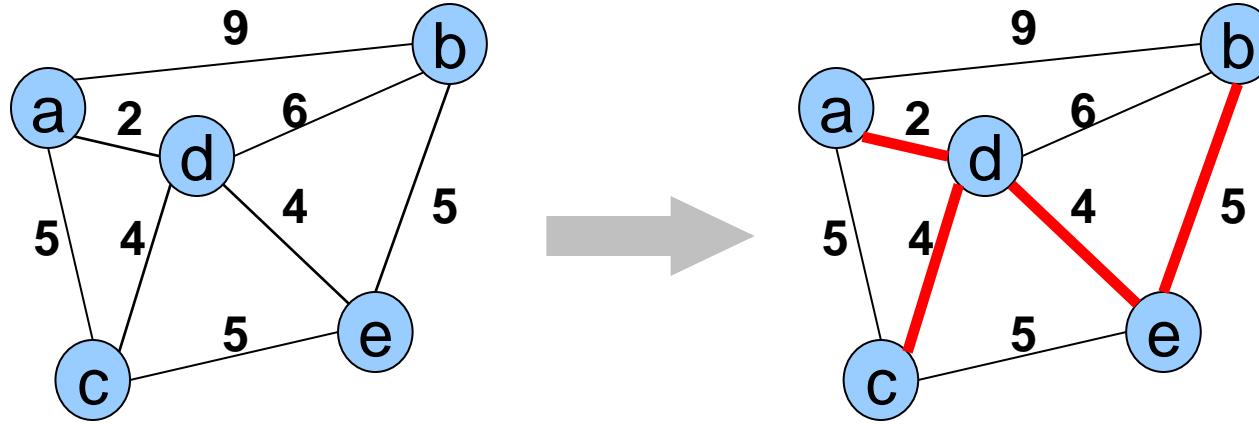
If Nodes represent cities and edges represent possible communication links between cities, then the minimum number of links required to connect all n cities is given by $n-1$ edges.

Example: Road network (Curtosy Mukund Madhavan CMI)

District hit by a cyclone, damaging the roads. Government sets to work to restore the roads. Priority is to ensure that all parts of the district can be reached

Which set of roads should be restored first?

How Can We Generate a MST?



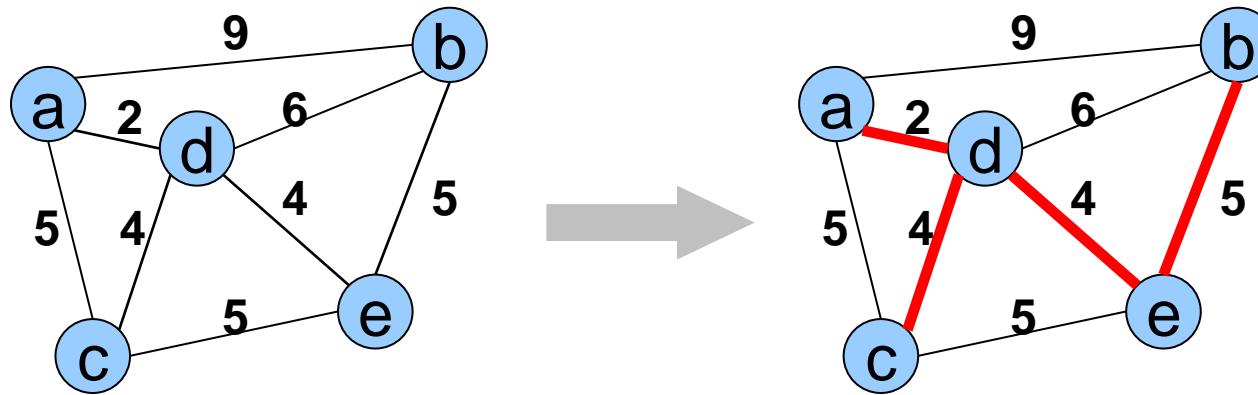
Generic approach:

A tree is an acyclic graph.

The idea is to start with an empty graph and try to add edges one at a time, always making sure that what is built remains acyclic.

And if we are sure every time the resulting graph always is a subset of some minimum spanning tree, then .. we are done.

Feasible Solutions



$$a-b-e-c-d = 9+5+5+4 = 23$$

$$a-c-d-b-e = 5+4+6+5 = 20$$

$$a-d-b-e-c = 2+6+5+5 = 18$$

$$a-d-c-e-b = 2+4+4+5 = 15 \dots\dots \text{Optimal}$$

Many Feasible solutions possible for the above Graph.

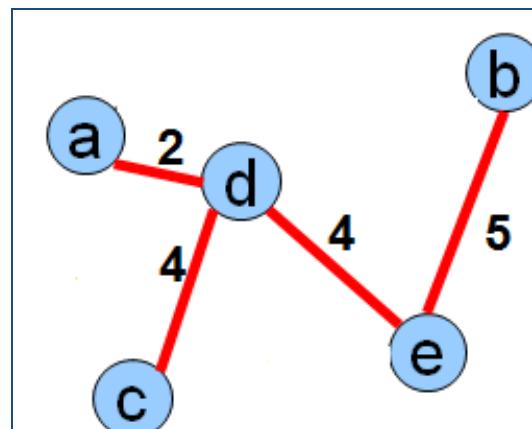
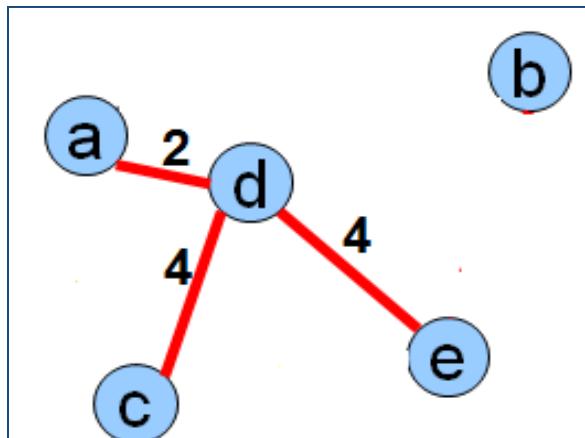
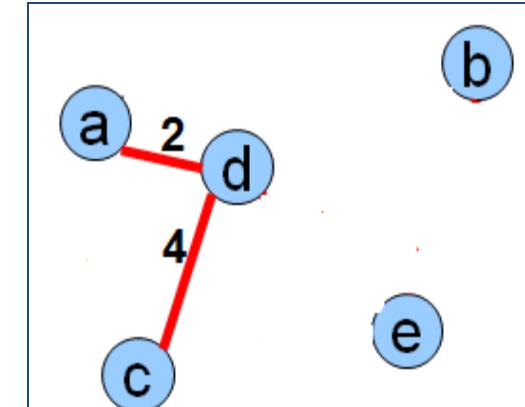
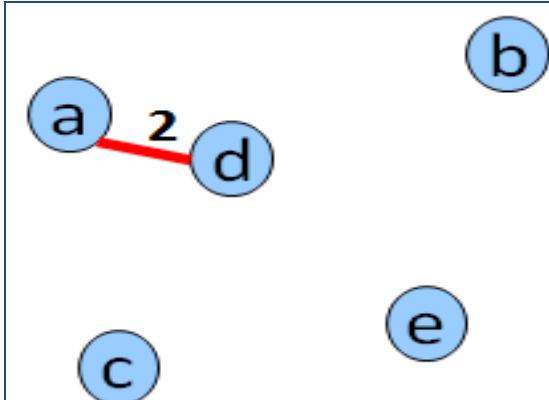
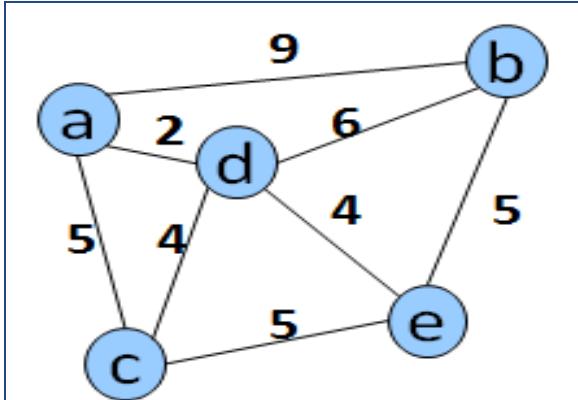
There are **two approaches** to derive an **optimal solutions**

1. Prims MST
2. Kruskals MST

Prims Minimum Cost Spanning Tree

- Builds min-cost spanning tree, edge by edge. Chooses an edge that results in minimum increase in the sum of the costs of the edges so far included.
- Start with a vertex u , select a minimum cost edge connecting vertex v .
- Now $u-v$ is the first edge selected into the tree.
- We continue to choose the edges into the solution set
 - In each iteration, we add one vertex(v) to the tree.
 - We choose an edge (u, v) such that u is a vertex in tree and v is a vertex not in tree and the edge cost (u, v) is minimum among all choices of vertices in tree to the vertices not in tree.

Prims Minimum Cost Spanning Tree Example



MST by Prims with Cost = 15

{a, d, c, e} select the
m select the cost edge
from all the edges
from {a, d, c, e}
cost edge get the
a-c = 5 → cycle,
reject edge
a-b = 9 → edge
c-e = 5 → cycle,
a-d = 2
reject
a-d = 2
d-b = 6 = 9
d-c = 4 = 5
e-b = 5 = 5
d-a-b = 9
d-e = 4
d-b = 6

```
1  Algorithm Prim( $E, cost, n, t$ )
2  //  $E$  is the set of edges in  $G$ .  $cost[1 : n, 1 : n]$  is the cost
3  // adjacency matrix of an  $n$  vertex graph such that  $cost[i, j]$  is
4  // either a positive real number or  $\infty$  if no edge  $(i, j)$  exists.
5  // A minimum spanning tree is computed and stored as a set of
6  // edges in the array  $t[1 : n - 1, 1 : 2]$ .  $(t[i, 1], t[i, 2])$  is an edge in
7  // the minimum-cost spanning tree. The final cost is returned.
8  {
9      Let  $(k, l)$  be an edge of minimum cost in  $E$ ;
10      $mincost := cost[k, l];$ 
11      $t[1, 1] := k; t[1, 2] := l;$ 
12     for  $i := 1$  to  $n$  do // Initialize near.
13         if ( $cost[i, l] < cost[i, k]$ ) then  $near[i] := l;$ 
14         else  $near[i] := k;$ 
15      $near[k] := near[l] := 0;$ 
```

```
16   for  $i := 2$  to  $n - 1$  do
17     { // Find  $n - 2$  additional edges for  $t$ .
18       Let  $j$  be an index such that  $near[j] \neq 0$  and
19        $cost[j, near[j]]$  is minimum;
20        $t[i, 1] := j$ ;  $t[i, 2] := near[j]$ ;
21        $mincost := mincost + cost[j, near[j]]$ ;
22        $near[j] := 0$ ;
23       for  $k := 1$  to  $n$  do // Update  $near[ ]$ .
24         if (( $near[k] \neq 0$ ) and ( $cost[k, near[k]] > cost[k, j]$ ))
25           then  $near[k] := j$ ;
26     }
27   return  $mincost$ ;
28 }
```

Prims time Complexity

To Insert $n-2$ vertices the Outer loop runs n times

With in the loop, O(n-2)

- In each iteration, we add one vertex(v) to the tree.
- We choose an edge (u, v) such that u is a vertex in tree and v is a vertex not in tree and the edge cost (u, v) is minimum among all choices of vertices in tree to the vertices not in tree.
 - To identify such an edge it takes $O(n)$ time. Near[] array is used for this in the algorithm. O(n)
 - After adding the edge, the near [] must be updated, which takes atmost $O(n)$ O(n)
- end loop

$$O(n-2) * (O(2n)) = O(2n^2 - 4n) = O(n^2)$$

hence Overall $O(n^2)$

Improvement in Prims time Complexity

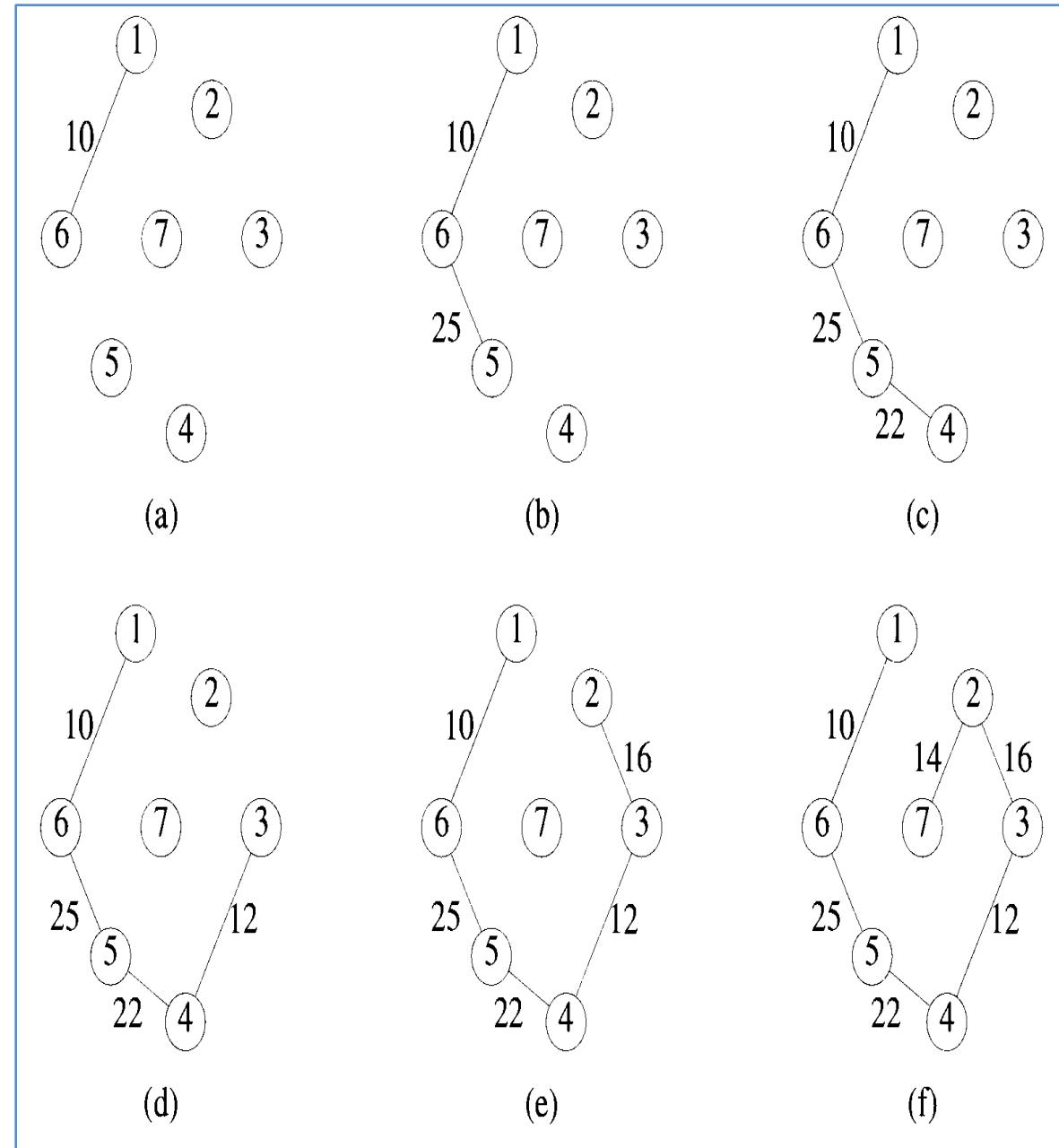
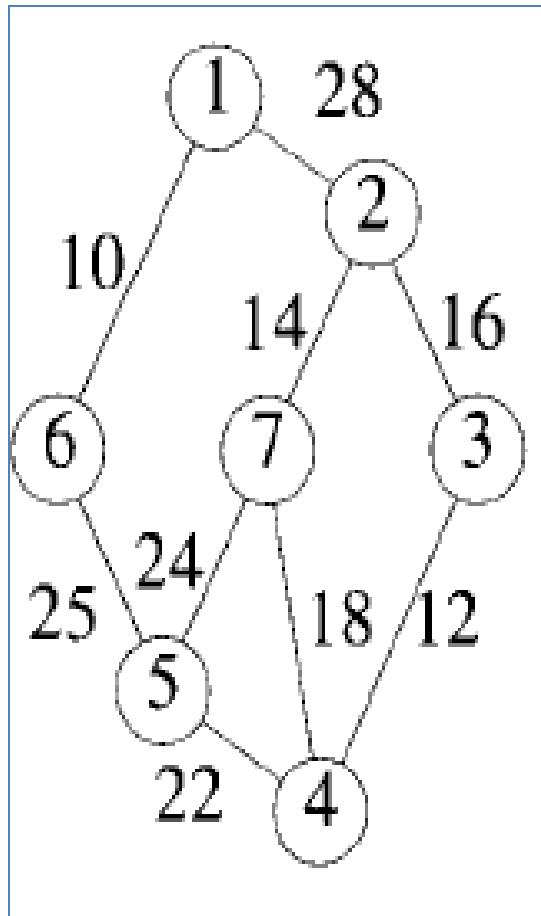
Instead of Adjacency matrix if Red-black trees are used then time complexity of **Prims** algorithm takes $O((n + |E|)\log n)$

If we store the nodes not yet included in the tree as a red-black tree instead of $O(n)$ it takes $O(\log n)$ time.

Note that a **red-black tree supports** the following operations in $O(\log n)$ time: **insert**, **delete**(an arbitrary element), **find-min**, and **search**(for an arbitrary element).

The for loop of line 23 has to examine only the nodes adjacent to j . Thus its overall frequency is $O(|E|)$, updating `near[]` takes $O(\log n)$ Thus the overall run time is $O(n + |E|) \log n$

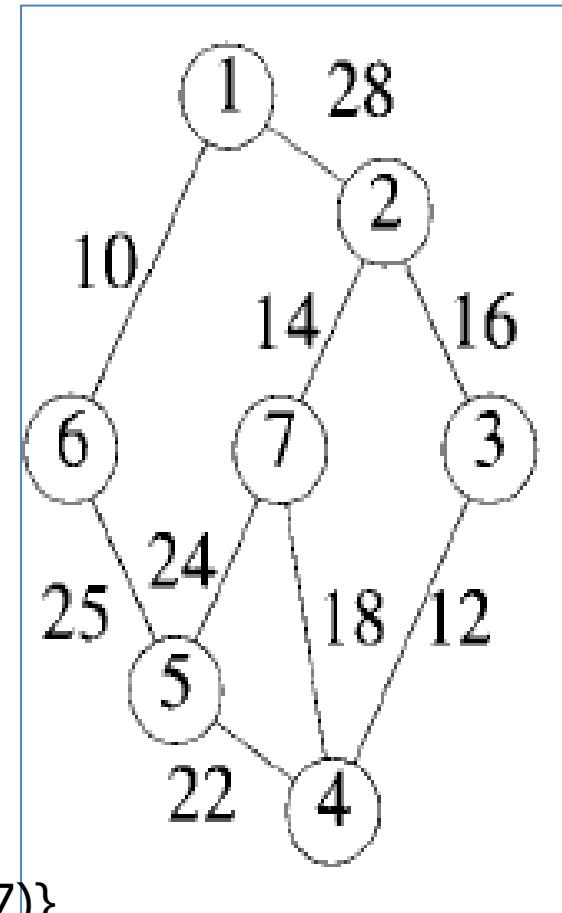
Tracing



Tracing Prims Algorithm

Cost adjacency matrix is :

	1	2	3	4	5	6	7
1	0	28	∞	∞	∞	10	∞
2	28	0	16	∞	∞	∞	14
3	∞	16	0	12	∞	∞	∞
4	∞	∞	12	0	22	∞	∞
5	∞	∞	∞	22	0	25	24
6	10	∞	∞	∞	25	0	∞
7	∞	14	∞	18	24	∞	0



$$E = \{ (1,6), (1,2), (6,5), (5,7), (5,4), (7,4), (4,3), (3,2), (2,7) \}$$

Initially select edge (1,6) as it is minimum cost edge in the Graph G.

Initialize **near array**

0	0	0	0	0	0	0
---	---	---	---	---	---	---

Tracing

I	Select Edge	K	L	Min Cost	T[I,k]	T[I,L]
1	(1,6)	1	6	10	1	6
2	(5,6)	6	5	10+25 =35	5	6
3	(5,4)	5	4	35+22 =57	5	4

Update near

and set near 1, 6 to 0

0	1	1	1	6	0	1
0	1	1	5	0	0	5

Tracing

j=1 near [1]==0 j=2 near[2] ≠ 0

cost[2, near[2]]= cost[2,1]= 28 j=3 near[3] ≠ 0

cost[3,near[3]]=cost [3,1]=∞ j=4 near[4] ≠ 0

cost[4,near[4]]=cost[4,1]= ∞ j=5 near[5] ≠ 0

cost[5,near[5]]=cost[5,6]=25 j=6 near[6]==0

j=7 near[7] ≠ 0

cost[7,near[7]]=cost[7,1]= ∞

so mincost is 25 & j=5 select (6,5) => t[2,1]=5; t[2,2]=near[5]=6

mincosif ((neart=10 +cost[5,6] = 10 +25=35

Set near[5]=0

for k=1 If((near[1] ≠ 0) && (cost[k,near[k]] > cost[k,j])) (0 ≠ 0)----false

k=2 if ((1 ≠ 0) && (28 > ∞)----false

k=3 if ((1 ≠ 0) && (∞ > ∞))----false

k=4 if ((1 ≠ 0) && (∞ > 22))----true near[4]=5

k=5 if (6==0) && (25 > ∞)---false

k=6 if ((0==0)----false

k=7 if (1 ≠ 0) && (∞ > 24)---true near[7]=5

Kruskal's MST

Initialization

1. Assume a set for each vertex $v \in V$

$$F = \{a\}, \{b\}, \{c\}, \{d\}, \{e\}$$

2. Sort edges in the increasing order of cost

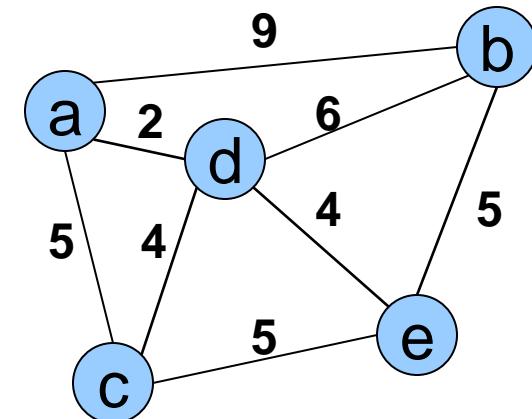
$$E = \{(a,d), (c,d), (d,e), (a,c), (b,e), (c,e), (b,d), (a,b)\}$$

3. Let T be the Tree of edges selected for MST.

initially T Comprising MST is empty set $T = \emptyset$

In each iteration Select a minimum cost edge (u, v) , such that u and v are vertices belonging to two different sets.

Connect the forest of tree



Kruskal's MST

1. Create a forest (a set of trees), where each vertex in the graph is in a separate tree.
 2. Arrange the edges in Ascending order of edge costs
 3. T is the spanning tree to be constructed.
- In each iteration select an edge (u, v) with min. cost
 - If this edge (u, v) belongs to two different trees (*ie. sets*) then connect the trees (**set union**), otherwise discard the edge.
 - Repeat the above steps till n vertices are connected

Kruskal's Algorithm

//E is the set of edges in G. Cost[u,v] is the cost of edge (u,v)

//t is the set of edges in MST, Final cost is returned

Algorithm(E, cost, n, t){

 construct a min-heap out of edge costs

 for i=1 to n do parent[i] = -1; //each vertex is in a different set.

 i=0; mincost=0;

 while((i<n-1) and (heap not empty)) do

{

 Delete a minimum cost edge (u,v) from the heap and
 re-heapify.

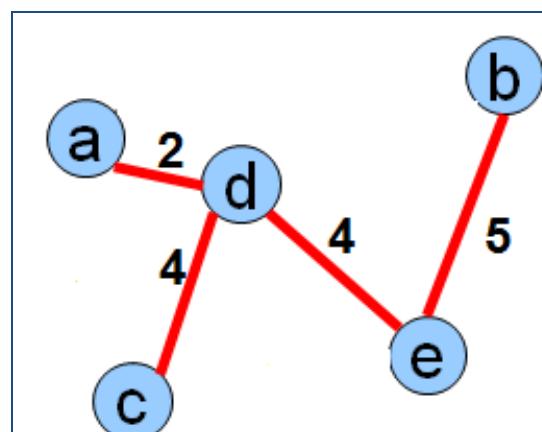
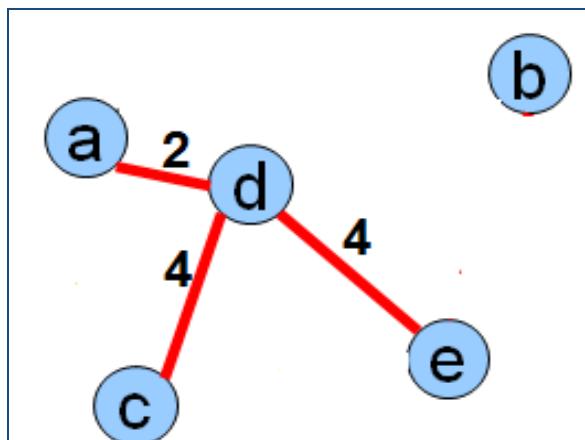
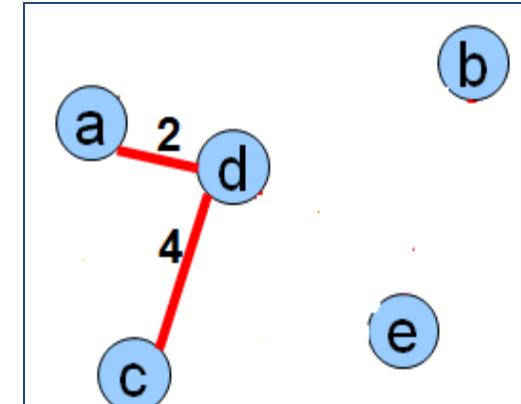
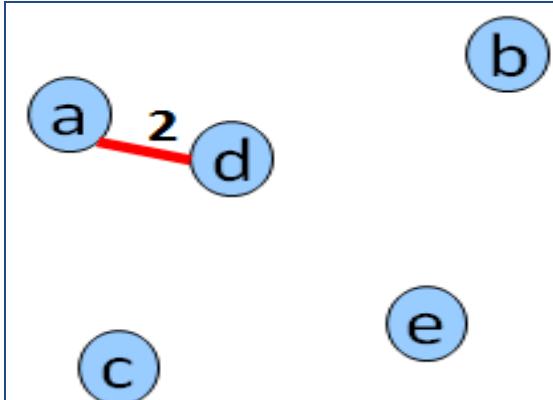
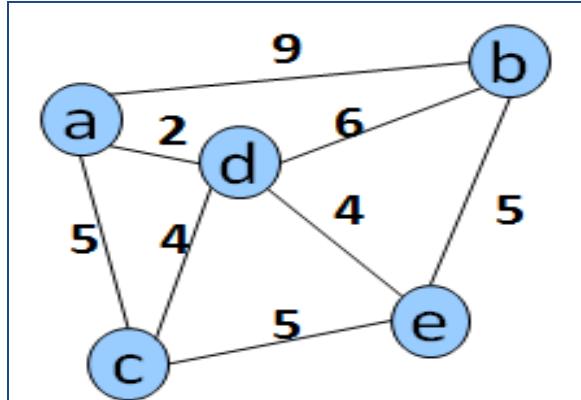
 j=Find(u); k=Find(v);

 // if (j==k) implies inclusion of edge(u,v) forms a cycle

Kruskal's Algorithm Conti..

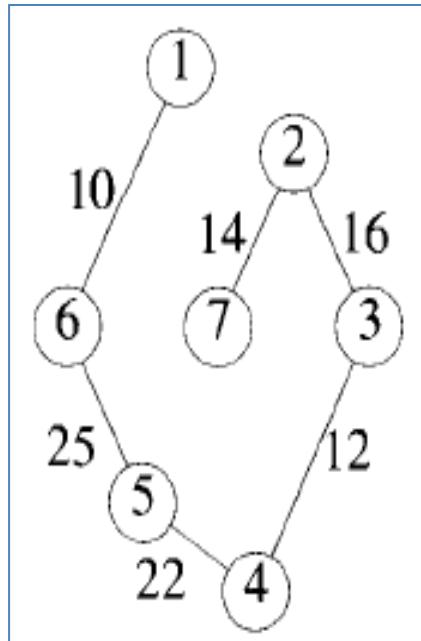
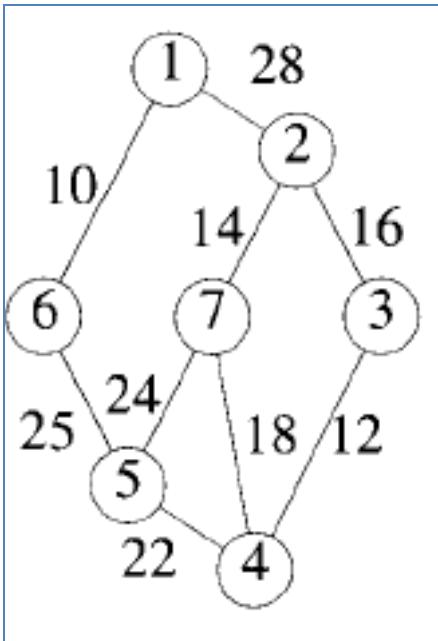
```
if(j!=k) then
{
    i++;
    t[l,1]= u; t[l,2]=v;
    mincost = mincost + cost[u,v];
    union(j,k);
}
If(i!=n-1) then write ("No spanning tree");
Else
    return mincost;
}
```

Kruskal Minimum Cost Spanning Tree Example



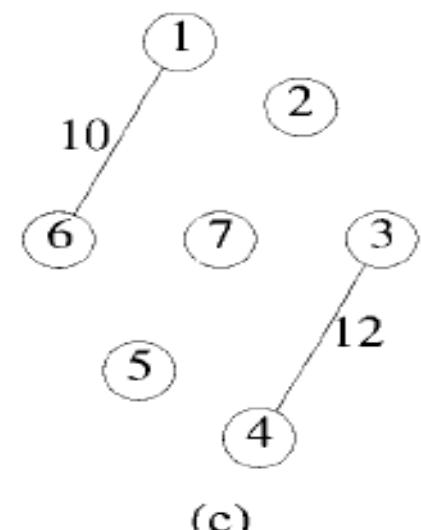
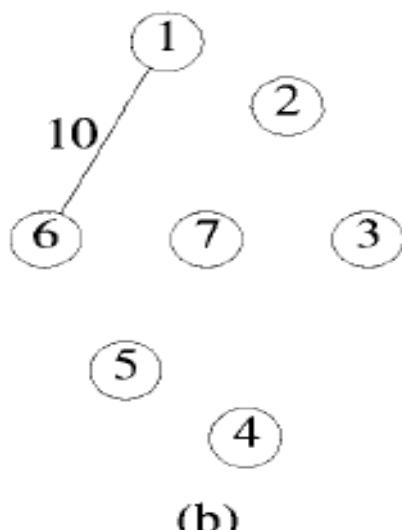
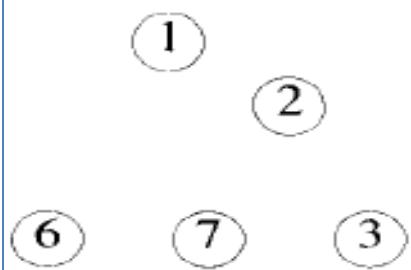
{ $a-d, d-e$ } select the
 $a-d = 2$
minimum cost edge
 $d-e = 4$ is selected
from all the edges
 $a-c = 5$
 $c-e = 4$
 $e-b = 5$
 $b-d = 6$
 $d-c = 6$
 $c-a = 6$
 $a-b = 9$ All the
 $a-d = 2$
 $a-c = 5$ > cycle,
 $d-c-a-c-e = 4$
 $e-b = 5$
 $b-d = 6$ is
 $d-e = 5$ > cycle,
 $e-d = 9$ or
 $d-selects d-e$
 $d-b = 6$
 $e-b \neq 5$
 $d-c$ be
selected.

MST by Prims with Cost = 15



Kruskal
Minimum Cost
Spanning Tree.

Refer Text book
for complete
solution.



(a)

(b)

(c)

Time Complexity

- Construction of Min Heap Requires $E \log E$
- **Within loop**
- Each time the next edge to be selected into MST is a minimum Cost edge
- Heap tree is constructed for arranging the edges in ascending order of edge cost
 - Each deletion operations of a Heap tree requires $\Theta(\log |E|)$.
 - Each edge selection into solution takes $\Theta(\log |E|)$.
 - $\Theta(E \log |E|)$ is the total time for construction of MST.

Activity Selection Problem

- It is the problem of scheduling several competing activities that require exclusive use of a common resource.
- The objective is to select a maximum-size set of mutually compatible activities.
- Suppose we have a set $S = \{a_1, a_2, \dots, a_n\}$ of n proposed activities that wish to use a resource, Each activity a_i has a start time s_i and a finish time f_i where $0 \leq s_i < f_i < \infty$.
- Activities a_i and a_j are compatible if the intervals s_i, f_i & s_j, f_j do not overlap.

Activity Selection Problem

- **Greedy approach:** We should choose an activity that leaves the resources available for as many other activities as possible.
- the activities are sorted in increasing order of finish time.
- **Example**

Consider the following 6 activities $\{a_1, a_2, \dots, a_6\}$ start[] = {5,1,3, 0,5,8}; finish[] = {9,2,4,6,7,9}; find maximum set of activities that can be executed .

Arrange the data in increasing order of finish time.

Time	a_2	a_3	a_4	a_5	a_1	a_6
Start	1	3	0	5	5	8
Finish	2	4	6	7	9	9

Activity

Time	a_2	a_3	a_4	a_5	a_1	a_6
Start	1	3	0	5	5	8
Finish	2	4	6	7	9	9

2 jobs:

(1,2) (3,4) -- a_2, a_3

(1,2) (5,7) -- a_2, a_5

(1,2) (8,9) -- a_2, a_6

(1,2) (5,9) -- a_2, a_1

(3,4) (5,7) -- a_3, a_5

(5,7) (8,9) -- a_5, a_6

(0,6) (8,9) -- a_4, a_6

3 Jobs:

(1,2) (3,4) (5,7)

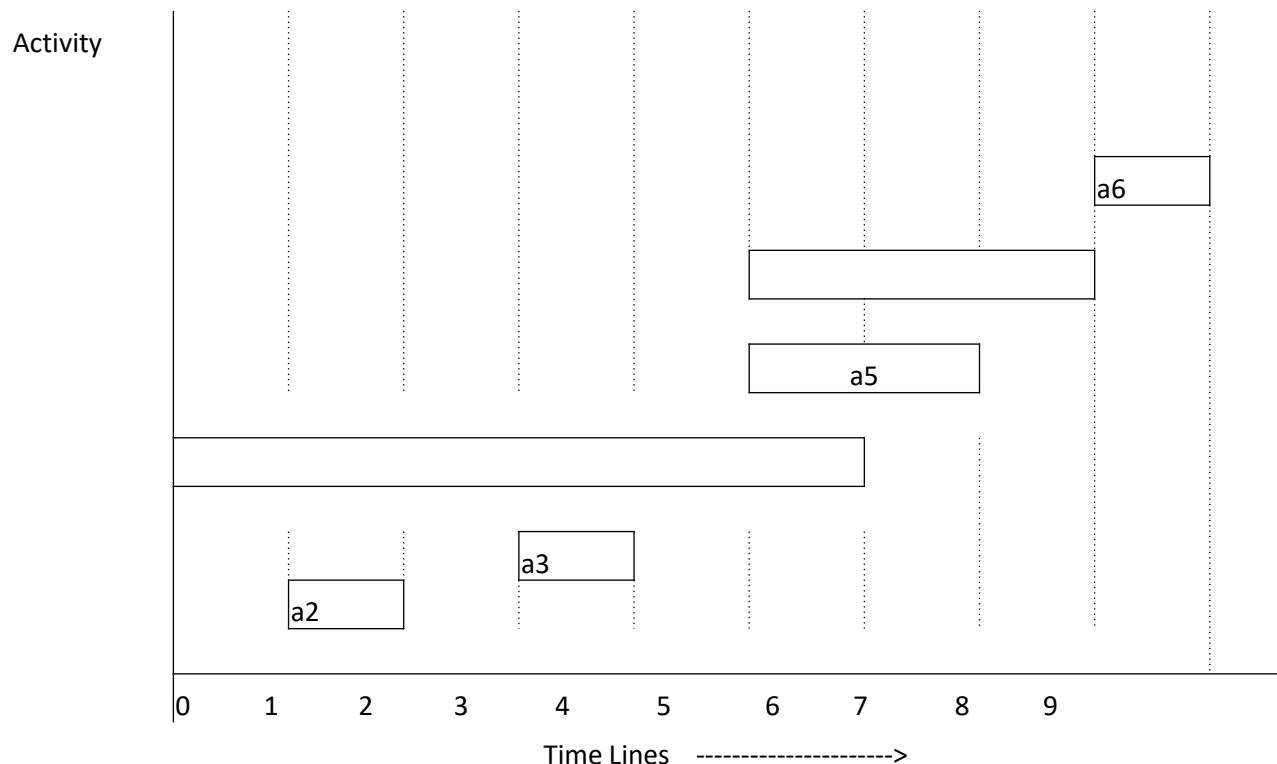
(1,2) (5,7) (8,9)

(3,4) (5,7) (8,9)

4 Jobs:

(1,2) (3,4) (5,7)(8,9)

Therefore, the optimal solution is : { a_2, a_3, a_5, a_6 } that is 4 activities can be executed.



Activity Selection Algorithm

GREEDY-ACTIVITY-SELECTOR (s, f)

```
1   $n = s.length$ 
2   $A = \{a_1\}$ 
3   $k = 1$ 
4  for  $m = 2$  to  $n$ 
5      if  $s[m] \geq f[k]$ 
6           $A = A \cup \{a_m\}$ 
7           $k = m$ 
8  return  $A$ 
```

Backtracking

- A technique to solve problems with a large search space, by systematically trying and **eliminating** possibilities that will not lead to **answer state**.
- The Problems which deal with searching for a set of solutions or which asks for an optimal solution, satisfying some constraints can be solved using backtracking.
- The basic idea of backtracking is to build up the solution vector one component at a time and to use modified criterion functions $P_i(x_1, \dots, x_i)$ called as **bounding functions** to test whether the vector being formed has any chances of success.
- The major **advantage** of this method is : if it is realized that the partial vector (x_1, x_2, \dots, x_i) can in **no way lead to** a solution state, then that **vector can be ignored**.

Backtracking

- Useful technique for optimizing search under complex set of constraints.
- Express the desired solution as an n-tuple (x_1, x_2, \dots, x_n) where each $x_i \in S_i$, S_i being a finite set
- The solution is based on finding one or more vectors that maximize, minimize, or satisfy a criterion function $P(x_1, x_2, \dots, x_n)$

Constraints

- Constraints are two types **implicit and explicit** Constraints.
- **Explicit constraints** are rules that restrict each x_i to take on values only from a given set S_i .
 - Depends on the instance I of problem being solved
 - All tuples that satisfy the explicit constraints define a solution space for I , **the values that x_i can be assigned**
 - Examples of explicit constraints,
 - $x_i \geq 0$, i.e, $S_i = \{\text{all nonnegative real numbers}\}$
 - $x_i = 0$ or 1
 - $l_i \leq x_i \leq u_i$
- **Implicit constraints** are rules that determine which of the tuples in the solution space of I satisfy the criterion function.
 - Implicit constraints describe the way in which the x_i 's must relate to each other.

Backtracking

- In Backtracking, the solution must be **feasible** and it may optimize an objective function
- We can represent the solution space for the problem using a **state space tree**.
 - State-Space Tree. A **state space tree** is a tree that represents all of the possible states of the problem, from the **root** as an initial state to the **leaf** as a terminal state.
 - The *root of the tree represents 0 choice*,
 - Nodes at depth 1 **represent** first choice
 - Nodes at depth 2 **represent** the second choice, etc.
 - In this tree a *path from a root to a leaf represents a candidate solution*

Backtracking

- If the Search of a SST leads to a feasible (or optimal) solution, then we say the **node is promising** otherwise a *non-promising state and exploration in this path may be discarded.*
- Backtracking is a DFS Search of the state space tree, with Bounding Function.
 - The function that kills the nodes that do not lead to answer state is Bounding Function.

N-queen problem- SS tree

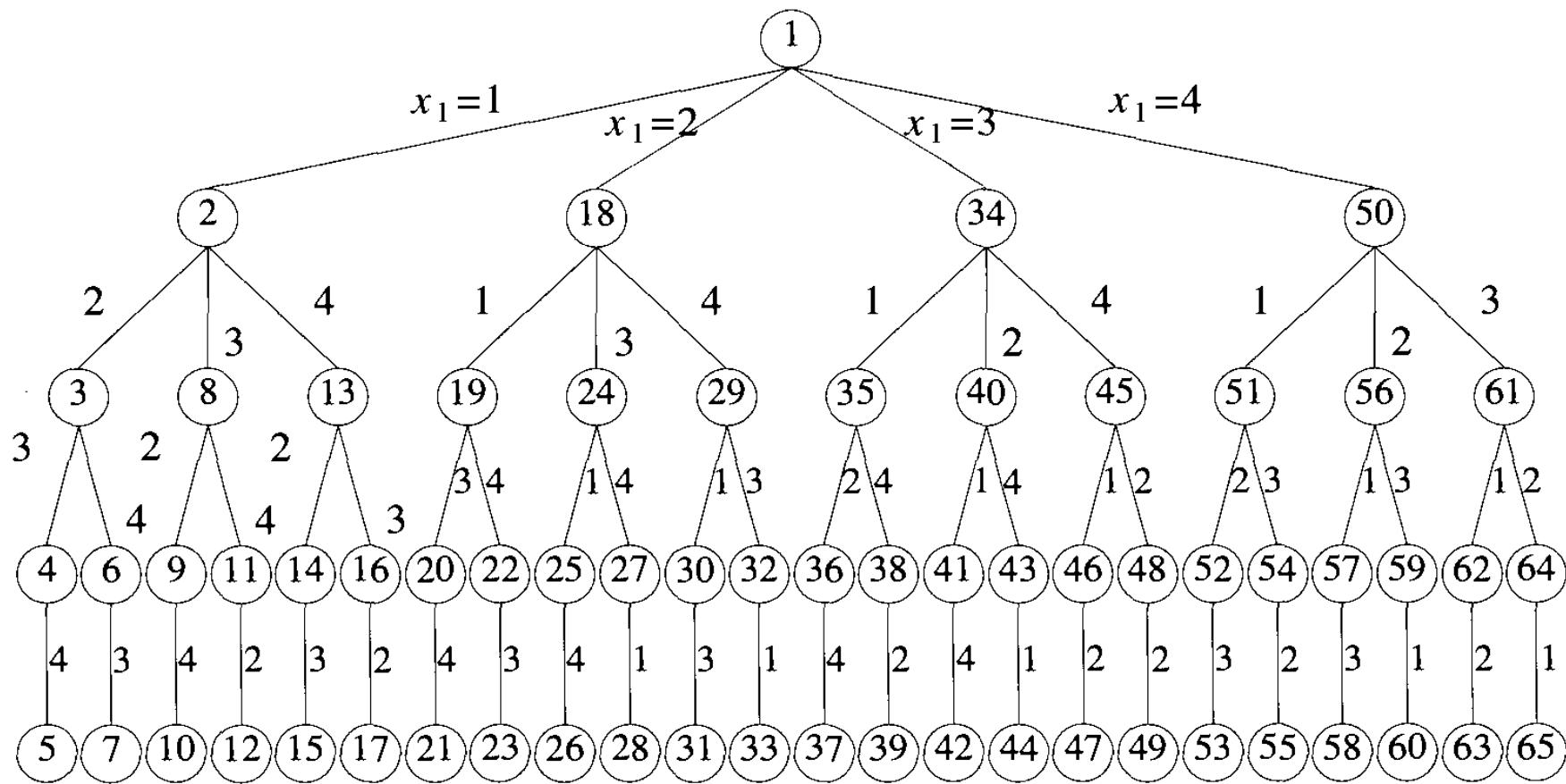


Figure 7.2 Tree organization of the 4-queens solution space. Nodes are numbered as in depth first search.

Sum of Subsets Problem

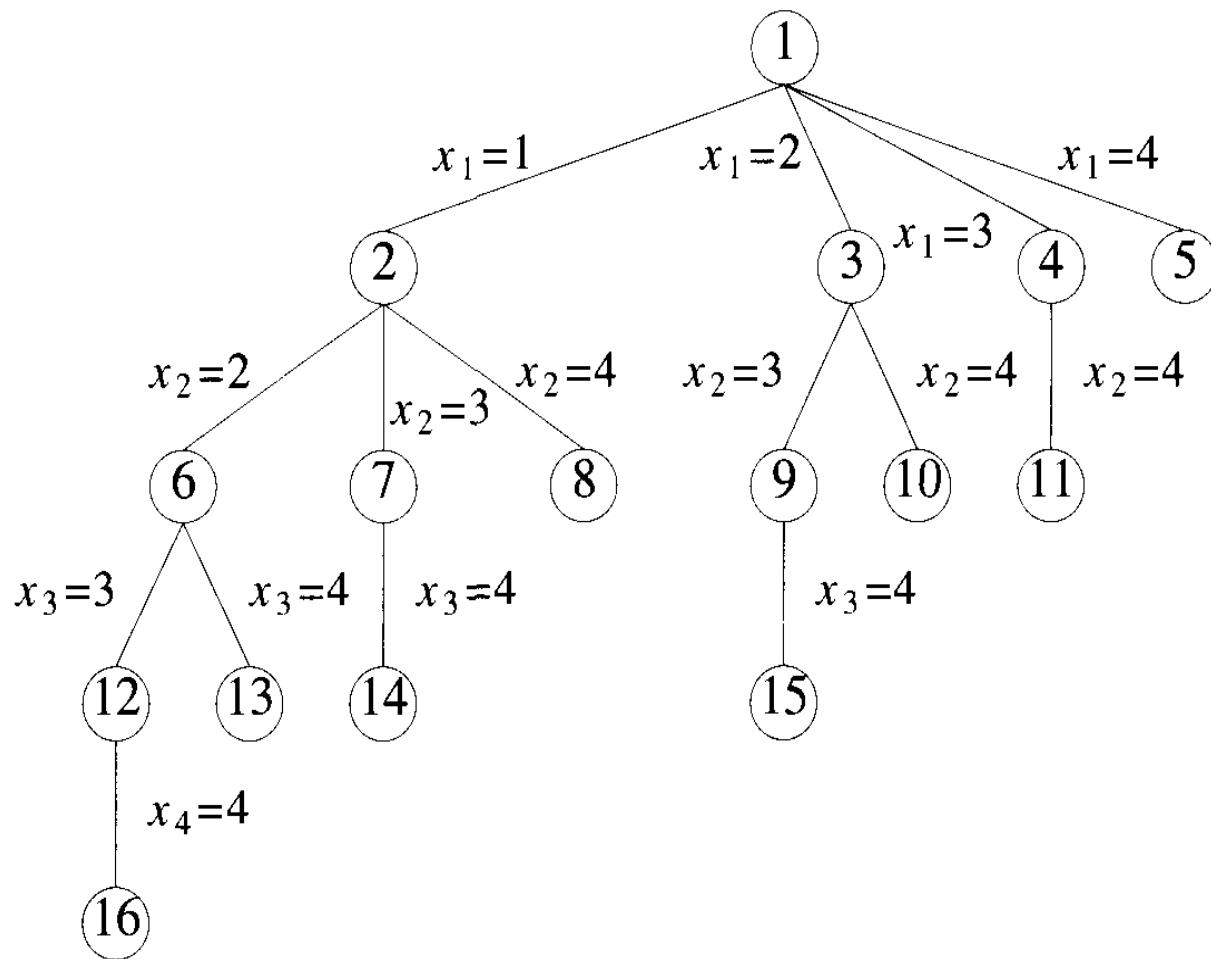


Figure 7.3 A possible solution space organization for the sum of subsets problem. Nodes are numbered as in breadth-first search.

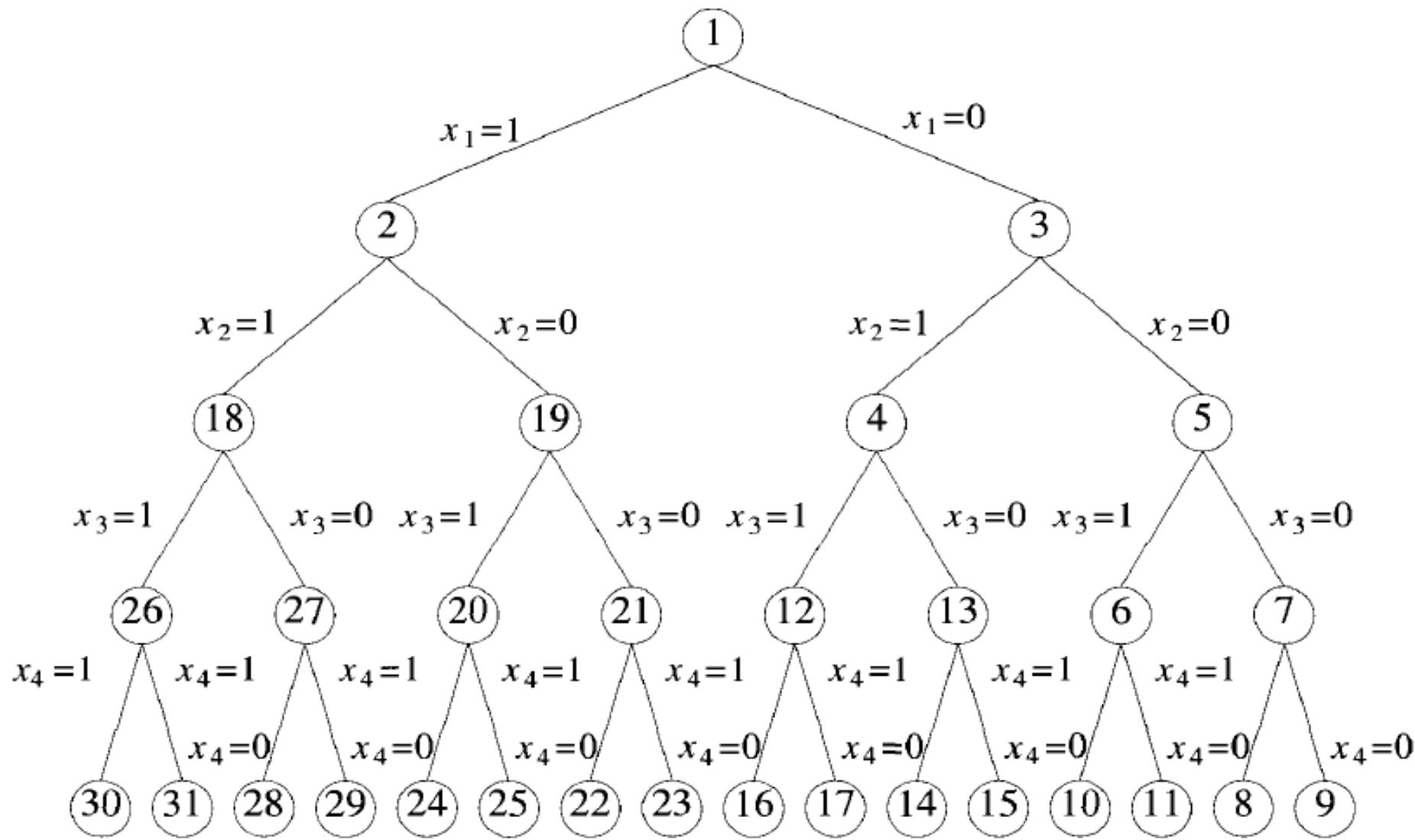


Figure 7.4 Another possible organization for the sum of subsets problems. Nodes are numbered as in D -search.

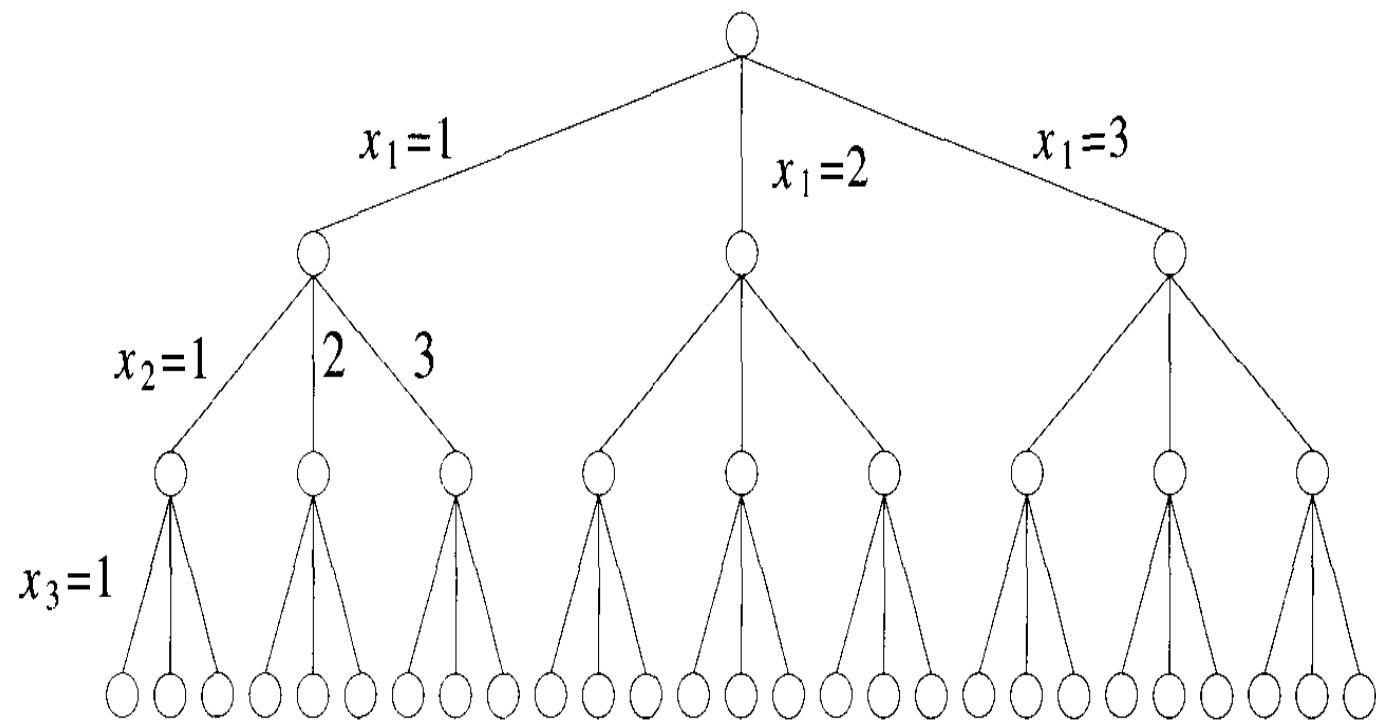


Figure 7.13 State space tree for mColoring when $n = 3$ and $m = 3$

State Space Tree

- Backtracking determines **problem solution** by systematically searching the solution space for the given problem instance
 - Solution Space is represented using a **State Space Tree (SST)**
- **State Space Tree Terminology**
 - **Problem state** : Each node in the tree represents the Problem state
 - **State space** : is the set of all paths from root to other nodes
 - **Solution states** : are those problem states **S** for which the path from the root to **S** defines a tuple in the solution space
 - In variable tuple size formulation tree, all nodes are solution states
 - In fixed tuple size formulation tree, only the leaf nodes are solution states
 - **Answer states** are those solution states **S** for which the path from root node to **S** defines a tuple that is a member of the set of solutions (ie it satisfies the implicit constraints)

State Space Tree

- **Live node** is a generated node for which all of the children have not been generated yet
- **E-node** is a live node whose children are currently being generated or explored
- **Dead node** is a generated node that is not to be expanded any further
 - All the children of a dead node are already generated
 - Live nodes are killed using a **bounding function** to make them dead nodes
- **Depth-first search**
 - As soon as a new child C of the current E-node P is generated, C becomes the new E-node; P becomes the E-node again when the subtree for C is fully explored
- **Bounding Functions** : Kill those live nodes (which do not lead to answer state) to make them dead nodes.
- **Backtracking is depth-first node generation with bounding functions.**

State Space Tree

- State space tree is the tree organization of the solution space , ie., every element of the solution space be represented by atleast one node in the SST
 - Static trees are ones for which tree organizations are independent of the problem instance being solved
 - Dynamic trees are ones for which organization is dependent on problem instance
- After conceiving state space tree for any problem,
 - The problem can be solved by systematically generating problem states
 - Checking which of them are solution states, and
 - Checking which solution states are answer states

Backtracking

- Brute force approach
 - Suppose m_i is the size of set S_i be
 - There are $m = m_1, m_2, \dots, m_n$, n-tuples that satisfy the criterion function P
 - In Brute force, you have to form **all the m n-tuples** to determine **the optimal solutions** by evaluating against P .
- Backtracking
 - Requires **less than m trials** to determine the solution
 - Form a solution (partial vector) one component at a time, and check at every step if this has any chance of success
 - If the solution **at any point seems not-promising**, ignore it
 - If the partial vector (x_1, x_2, \dots, x_i) do not yield a solution, ignore $m_{i+1} \dots m_n$ possible test vectors even without looking at them.

Backtracking Control Abstraction

- Assume that all answer nodes are to be found and not just one
- Let (x_1, x_2, \dots, x_i) be the path from root to a node in the state space tree
- Let $T(x_1, x_2, \dots, x_{i+1})$ be the set of all possible values for x_{i+1} such that $(x_1, x_2, \dots, x_{i+1})$ is also a path to a problem state
- $T(x_1, x_2, \dots, x_n) = \emptyset$;
- Assume a bounding function B_{i+1} expressed as a predicate such that if $B_{i+1}(x_1, x_2, \dots, x_{i+1})$ is false for a path $(x_1, x_2, \dots, x_{i+1})$ from root to a problem state, then the path cannot be extended to reach an answer node
- Thus, the candidate for position $i+1$ of the solution vector (x_1, x_2, \dots, x_n) are those values that are generated by T and satisfy B_{i+1} .

Recursive Backtracking – Control Abstraction

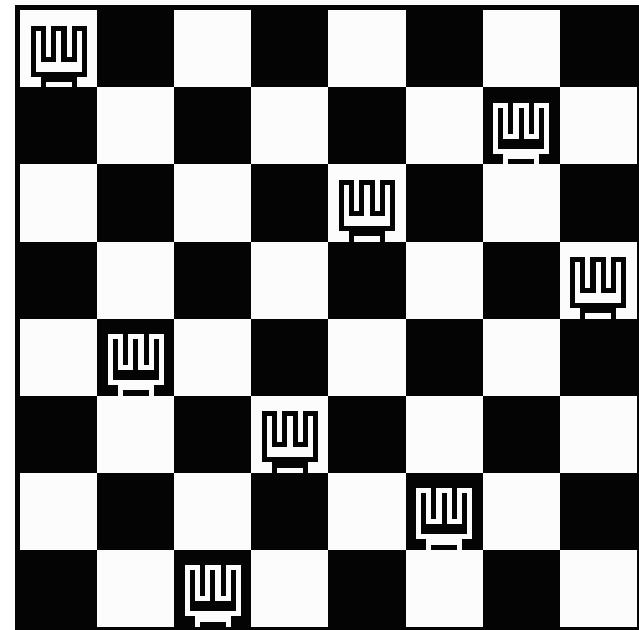
```
1 Algorithm Backtrack( $k$ )
2 // This schema describes the backtracking process using
3 // recursion. On entering, the first  $k - 1$  values
4 //  $x[1], x[2], \dots, x[k - 1]$  of the solution vector
5 //  $x[1 : n]$  have been assigned.  $x[ ]$  and  $n$  are global.
6 {
7     for (each  $x[k] \in T(x[1], \dots, x[k - 1])$ ) do
8     {
9         if ( $B_k(x[1], x[2], \dots, x[k]) \neq 0$ ) then
10        {
11            if ( $x[1], x[2], \dots, x[k]$  is a path to an answer node)
12                then write ( $x[1 : k]$ );
13            if ( $k < n$ ) then Backtrack( $k + 1$ );
14        }
15    }
16 }
```

Iterative Backtracking – Control Abstraction

```
1 Algorithm |Backtrack( $n$ )
2 // This schema describes the backtracking process.
3 // All solutions are generated in  $x[1 : n]$  and printed
4 // as soon as they are determined.
5 {
6      $k := 1$ ;
7     while ( $k \neq 0$ ) do
8     {
9         if (there remains an untried  $x[k] \in T(x[1], x[2], \dots,$ 
10             $x[k - 1])$  and  $B_k(x[1], \dots, x[k])$  is true) then
11         {
12             if ( $x[1], \dots, x[k]$  is a path to an answer node)
13                 then write ( $x[1 : k]$ );
14              $k := k + 1$ ; // Consider the next set.
15         }
16         else  $k := k - 1$ ; // Backtrack to the previous set.
17     }
18 }
```

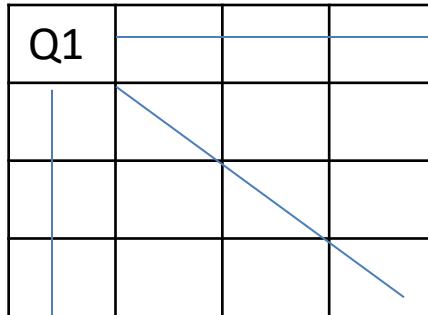
Backtracking – Eight Queens Problem

- Find an arrangement of 8 queens on a single chess board such that no two queens are attacking one another.
- In chess, queens can move all the way down any row, column or diagonal (so long as no pieces are in the way).
 - Due to the first two restrictions, it's clear that each row and column of the board will have exactly one queen.



nQueen

- All solutions to N-queen problem can be represented as N-tuples, (x_1, x_2, \dots, x_N) where x_i is the column on which queen i is placed.



Chess board Queen
Moves **horizontally**,
Vertically and
diagonally

- **Explicit constraints are**

$S_i = \{1, 2, 3, \dots, N\}$, the valid values that each x_i can take.

One of the possible candidate solution of x_i 's 4-Queen (2,4,1,3)

Each x_i can take value between 1-4.

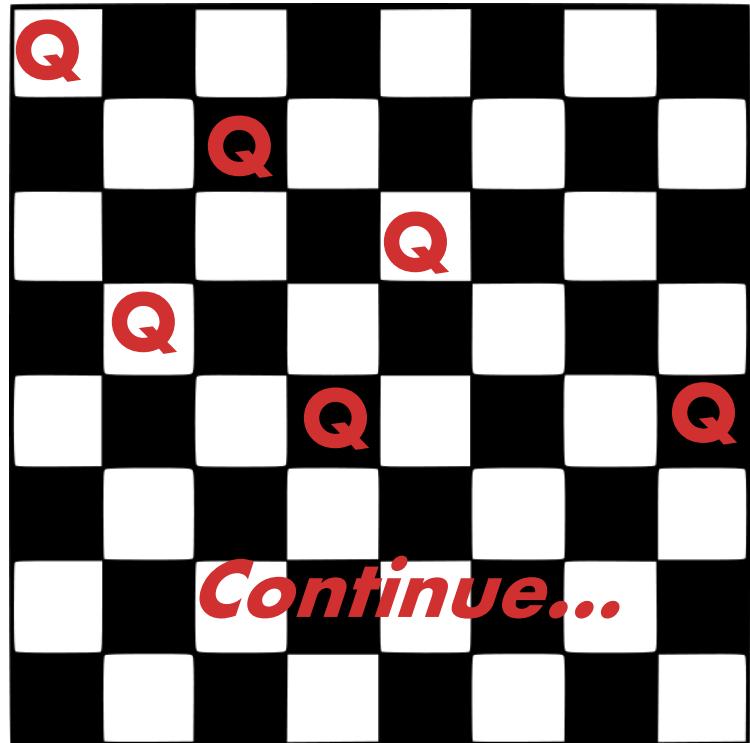
- **Implicit constraints are**

No two queens should be on same row and/or same column ie no two x_i 's should be same.

No two queens should attack each other diagonally, columnar

Backtracking – Eight Queens Problem

- The backtracking strategy is as follows:
 - 1) Place a queen on the first available square in row 1.
 - 2) Move onto the next row, placing a queen on the first available square there (that doesn't conflict with the previously placed queens).
 - 3) Continue in this fashion until either:
 - a) you have solved the problem, or
 - b) you get stuck.
 - When you get stuck, remove the queens that got you there, until you get to a row where there is another valid square to try.



Animated Example:
http://www.hbmeyer.de/backtrac_k/achtdamen/eight.htm#up

N-queen problem- SS tree

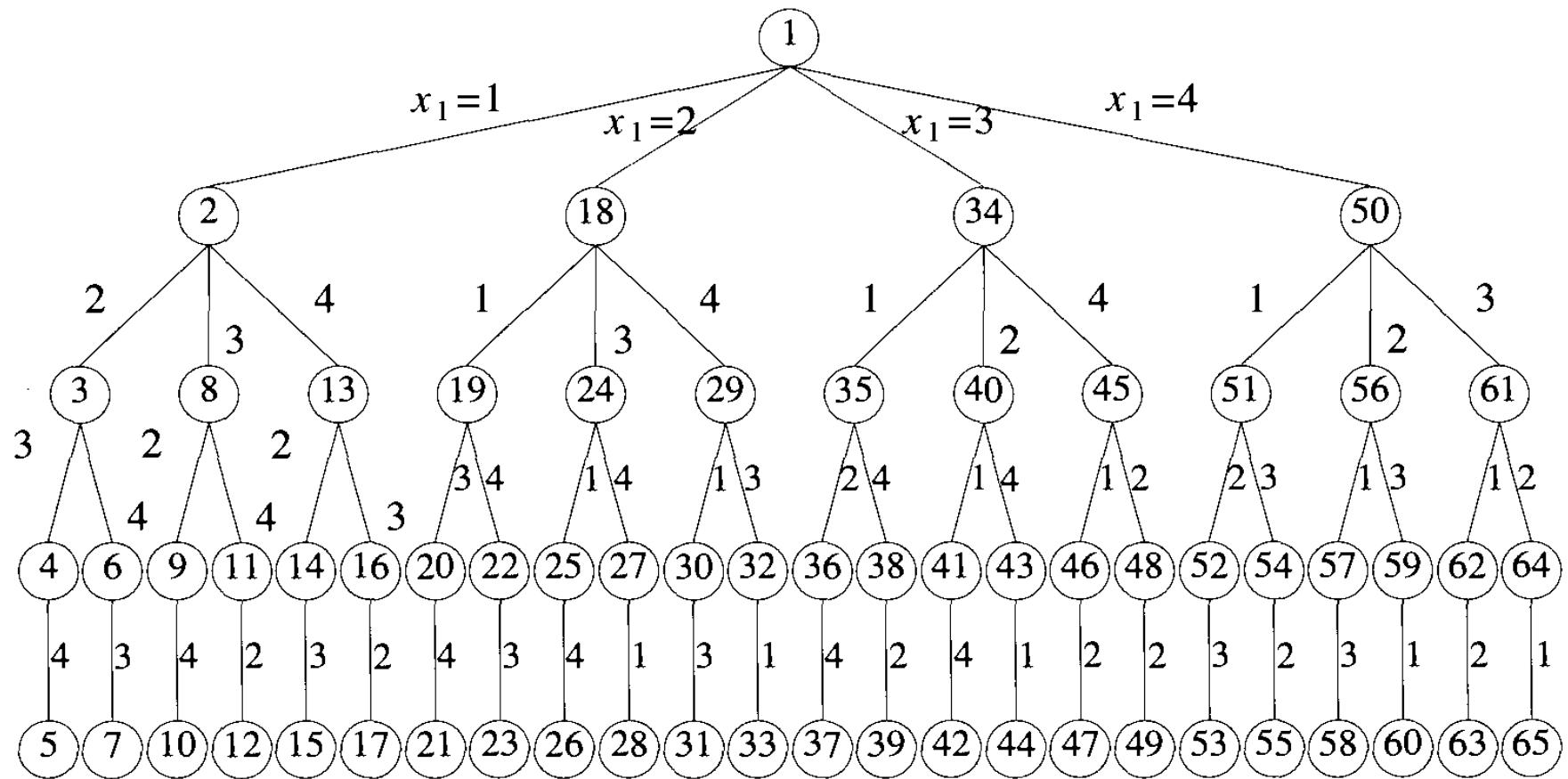


Figure 7.2 Tree organization of the 4-queens solution space. Nodes are numbered as in depth first search.

4-Queen Portion of Backtracking SST

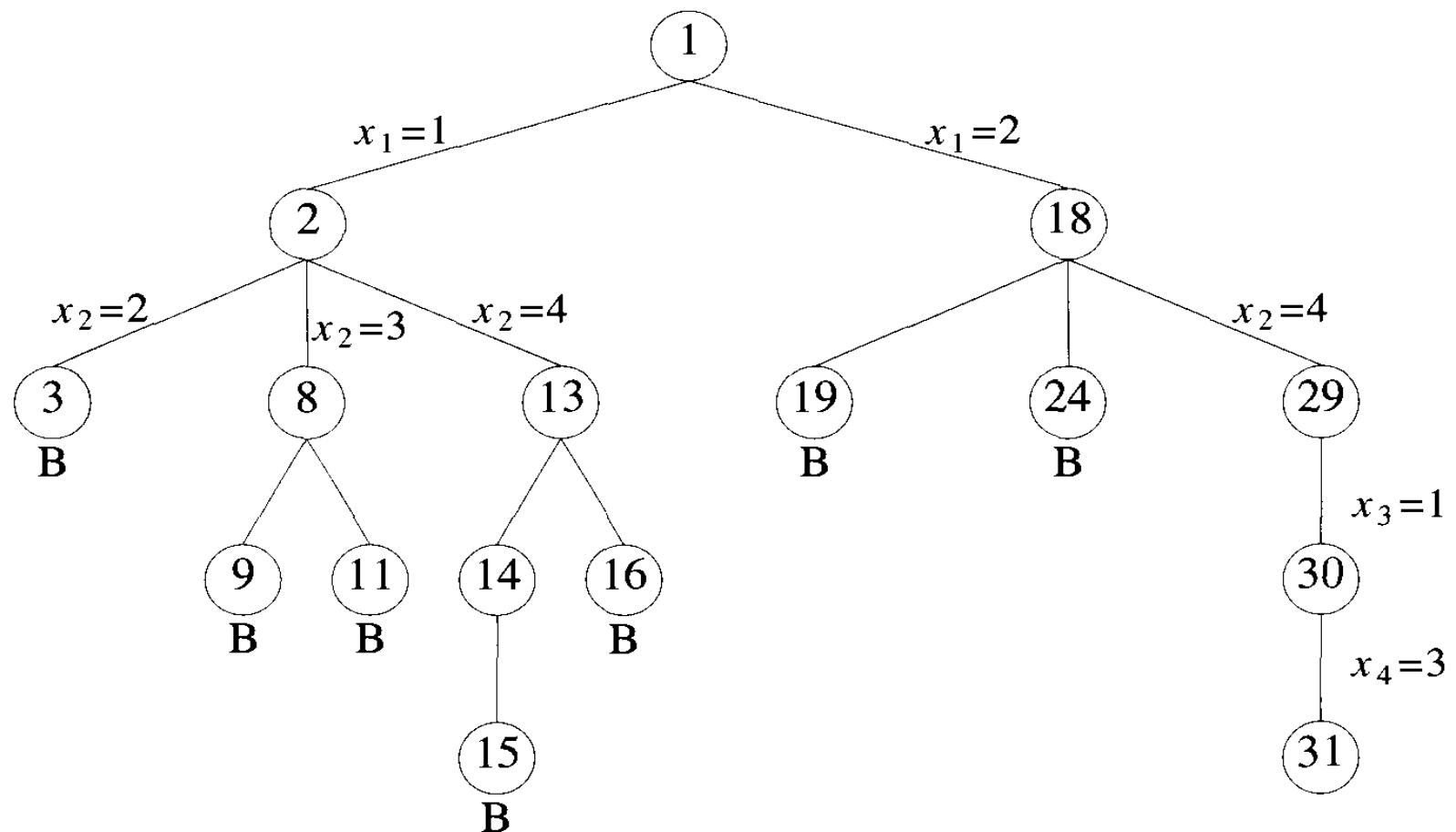


Figure 7.6 Portion of the tree of Figure 7.2 that is generated during backtracking

NQueens

```
1  Algorithm NQueens( $k, n$ )
2    // Using backtracking, this procedure prints all
3    // possible placements of  $n$  queens on an  $n \times n$ 
4    // chessboard so that they are nonattacking.
5    {
6      for  $i := 1$  to  $n$  do
7        {
8          if Place( $k, i$ ) then
9            {
10               $x[k] := i;$ 
11              if ( $k = n$ ) then write ( $x[1 : n]$ );
12              else NQueens( $k + 1, n$ );
13            }
14        }
15    }
```

Place – Can a new queen be placed

```
1  Algorithm Place( $k, i$ )
2    // Returns true if a queen can be placed in  $k$ th row and
3    //  $i$ th column. Otherwise it returns false.  $x[ ]$  is a
4    // global array whose first  $(k - 1)$  values have been set.
5    //  $\text{Abs}(r)$  returns the absolute value of  $r$ .
6    {
7      for  $j := 1$  to  $k - 1$  do
8        if  $((x[j] = i) \text{ // Two in the same column}$ 
9          or  $(\text{Abs}(x[j] - i) = \text{Abs}(j - k)))$ 
10         // or in the same diagonal
11         then return false;
12       return true;
13     }
```

Time Complexity

- The efficiency of backtracking algorithms depends on 4 factors.
 - Time to generate x_k .
 - No. of x_k satisfying explicit constraints
 - Time for Bounding function B_k .
 - The no. of x_k satisfying the B_k . -- **depends on pb inst**
- Bounding functions are regarded good, if they substantially reduce the no. of nodes that are generated.
- Once the SST organization is selected, 4th factor, **the no of nodes generation**, varies with the problem instance.
 - For some pb $O(n)$ nodes get generated
 - For some almost all the nodes in SST get generated.
- If the no. of nodes in SS is 2^n or $n!$ then the Worst Case time of Backtracking alg is $O(p(n) 2^n)$ or $O(q(n)n!)$, where $p(n), q(n)$ are polynomial time

Relatively
Independent of
Problm
Instnce

Estimate no. of nodes Generation

- Estimate the no. of nodes that get generated by BT algorithm using Monte Carlo method.
- The estimation is required particularly when all answer nodes are to be searched.
- The idea is to generate a random path in the SST.
 - Let X be a node on level i of SST,
 - BFs are used at node X to determine no. m_i of its child nodes that do not get bounded.
 - Next node on path is obtained by randomly selecting one of these m_i children that do not get bounded.
 - The path generation terminates at leaf or all of whose children are bounded.
 - Using these m_i 's we can estimate the total no. of nodes in the SST that will not be bounded.

Estimate no. of nodes Generation

- To estimate we assume two
 - static bounding function
 - nodes on same level have same degree
 - Let the expected no. of unbounded nodes on level 2 be m_1
 - Let m_2 be the no. of unbounded child nodes for each node at level 2. This yields total of $m_1 m_2$ nodes on level 3.
 - Let expected no. of unbounded Child nodes on level 4 be given by $m_1 m_2 m_3$.

=> Level 2 m_1
Level 3 $m_1 m_2$
Level 4 $m_1 m_2 m_3$
Level $i+1$ $m_1 m_2 m_3 \dots m_i$

The estimated no. of unbounded nodes in solving the problem
Instance I is given by $m = 1 + m_1 + m_1 m_2 + m_1 m_2 m_3 + \dots$

Analysis Nqueens

- If $N= 4$
- We require the examination of atmost $4!$ Possibilities by allowing only placements of queens on distinct rows and columns.
- The total no. of nodes in SST of 4 queen problem is given by
- $1 + \sum_{j=0}^{n-1} [\pi_{i=0}^j (n-i)]$

$$N=4$$

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

$$1 + 4 + 12 + 24 + 24 = 65$$

Sum of Subsets Problem

- Suppose we are given N distinct positive integers or numbers (w_i), where $1 \leq i \leq N$.
- Find subsets of these numbers that sum up to m.
- If $n=4$ (w_1, w_2, w_3, w_4)=(11,13,24,7) and $m=31$, then desired solutions are represented as
 - Variable Tuple Representation
 - $x_i \in \{j \mid j \text{ is an integer} \& 1 \leq j \leq N\}$.. Explicit Constraint
 - (11,13,7), (24,7) in terms of values of integers
 - (1,2,4), (3,4) in terms of index of Integers
 - Fixed Tuple Representation
 - $x_i \in \{0,1\}$.. Explicit Constraint
 - (1,1,0,1), (0,0,1,1)
 - Implicit Constraint
 - No two x_i be same
 - Sum of corresponding w_i s be m.

Sum of Subsets Problem

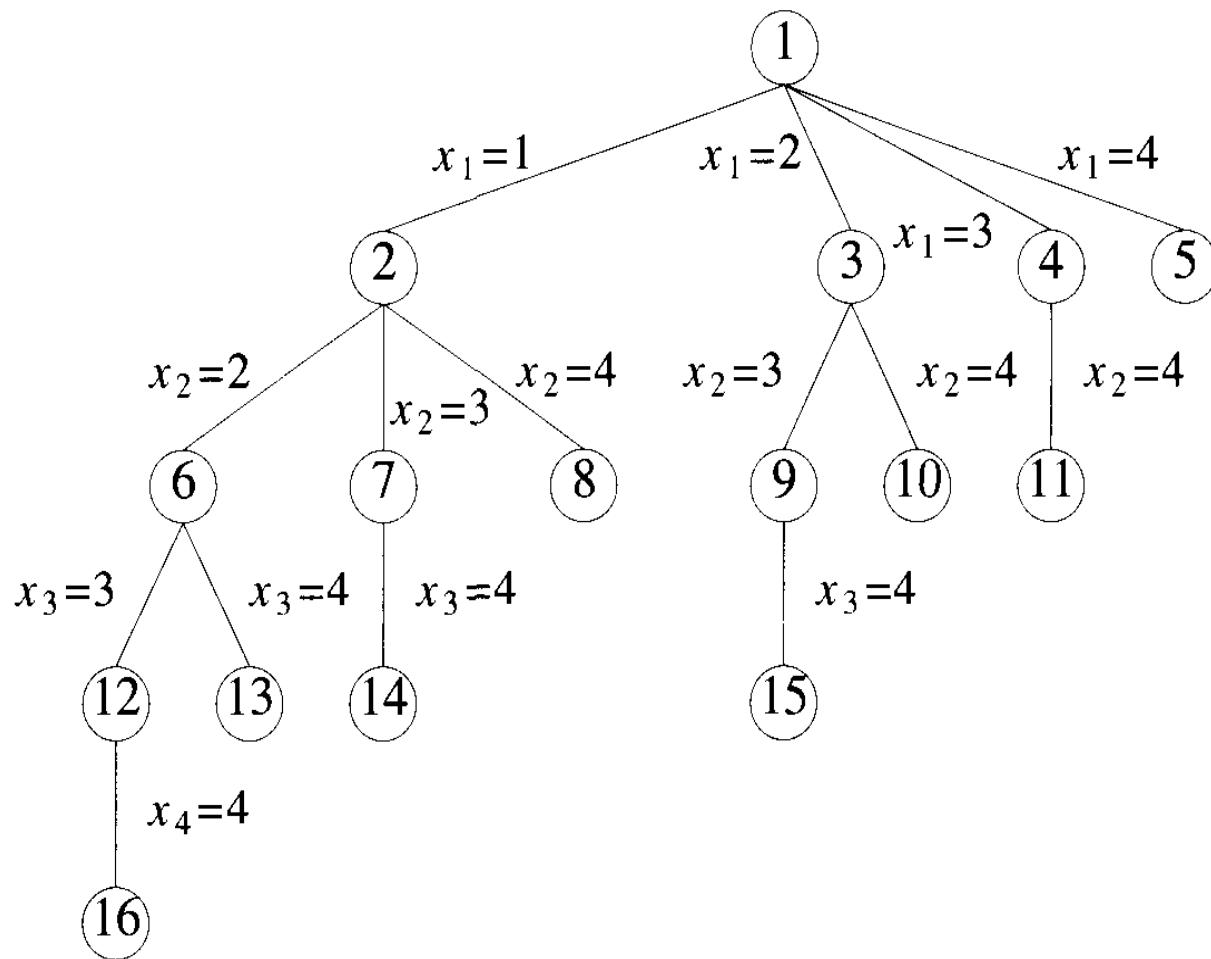


Figure 7.3 A possible solution space organization for the sum of subsets problem. Nodes are numbered as in breadth-first search.

Sum of Subsets Problem

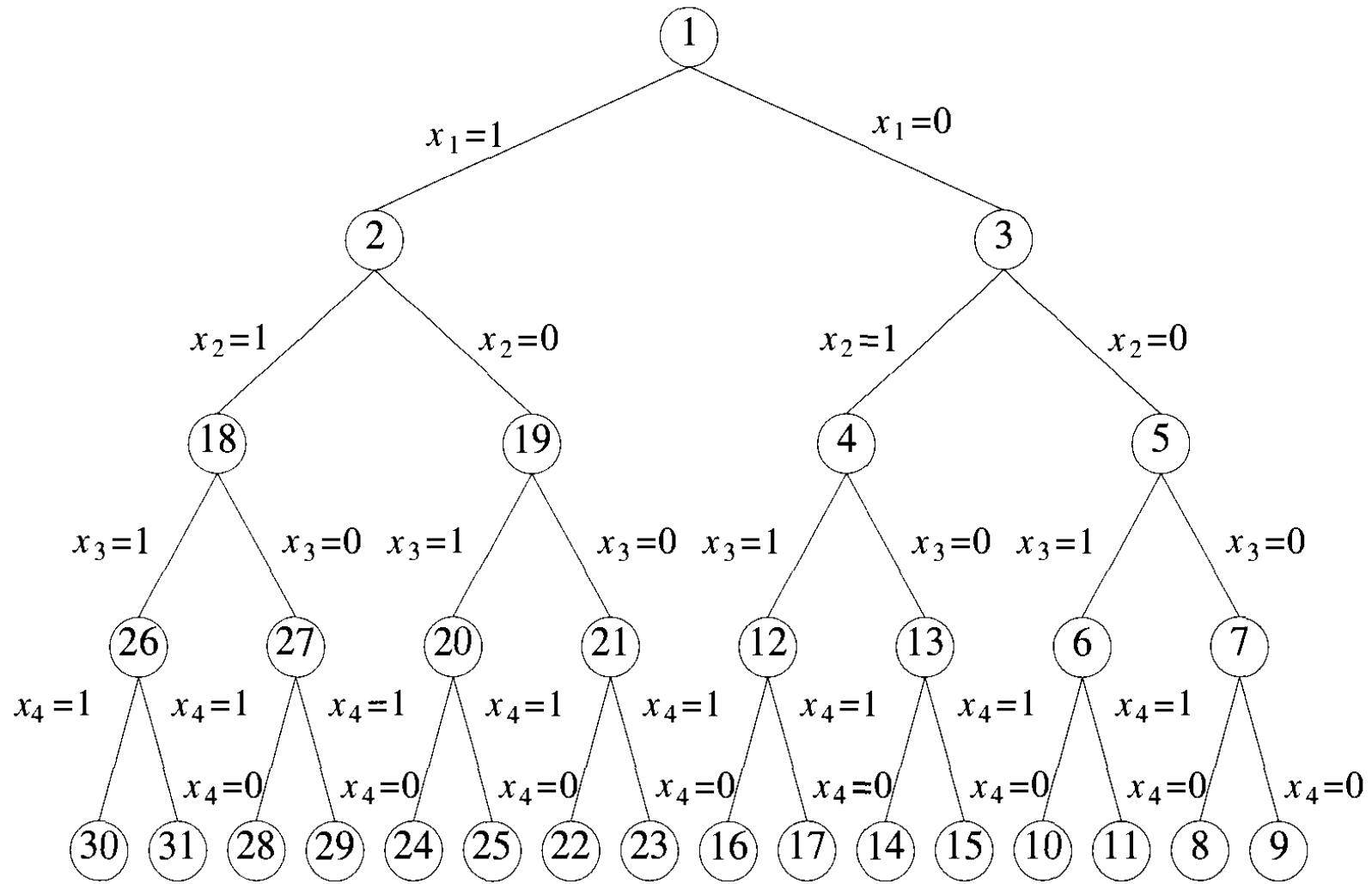


Figure 7.4 Another possible organization for the sum of subsets problems. Nodes are numbered as in D -search.

Bounding Function Sum of Subsets

$$\sum_{i=1}^k w_i x_i + \sum_{i=k+1}^n w_i \geq m$$

Total sum till now and remaining nos to be included do not sum up to m then there is no solution, **search need not proceed.**

$$\sum_{i=1}^k w_i x_i + w_{k+1} > m$$

Since Wi's are in increasing order. Inclusion of w_{k+1} in sum exceeds m then it cannot lead to solution, **search need not proceed.**

$$B_k(x_1, \dots, x_k) = \text{true iff } \sum_{i=1}^k w_i x_i + \sum_{i=k+1}^n w_i \geq m$$

$$\text{and } \sum_{i=1}^k w_i x_i + w_{k+1} \leq m$$

Sum of Subsets Problem

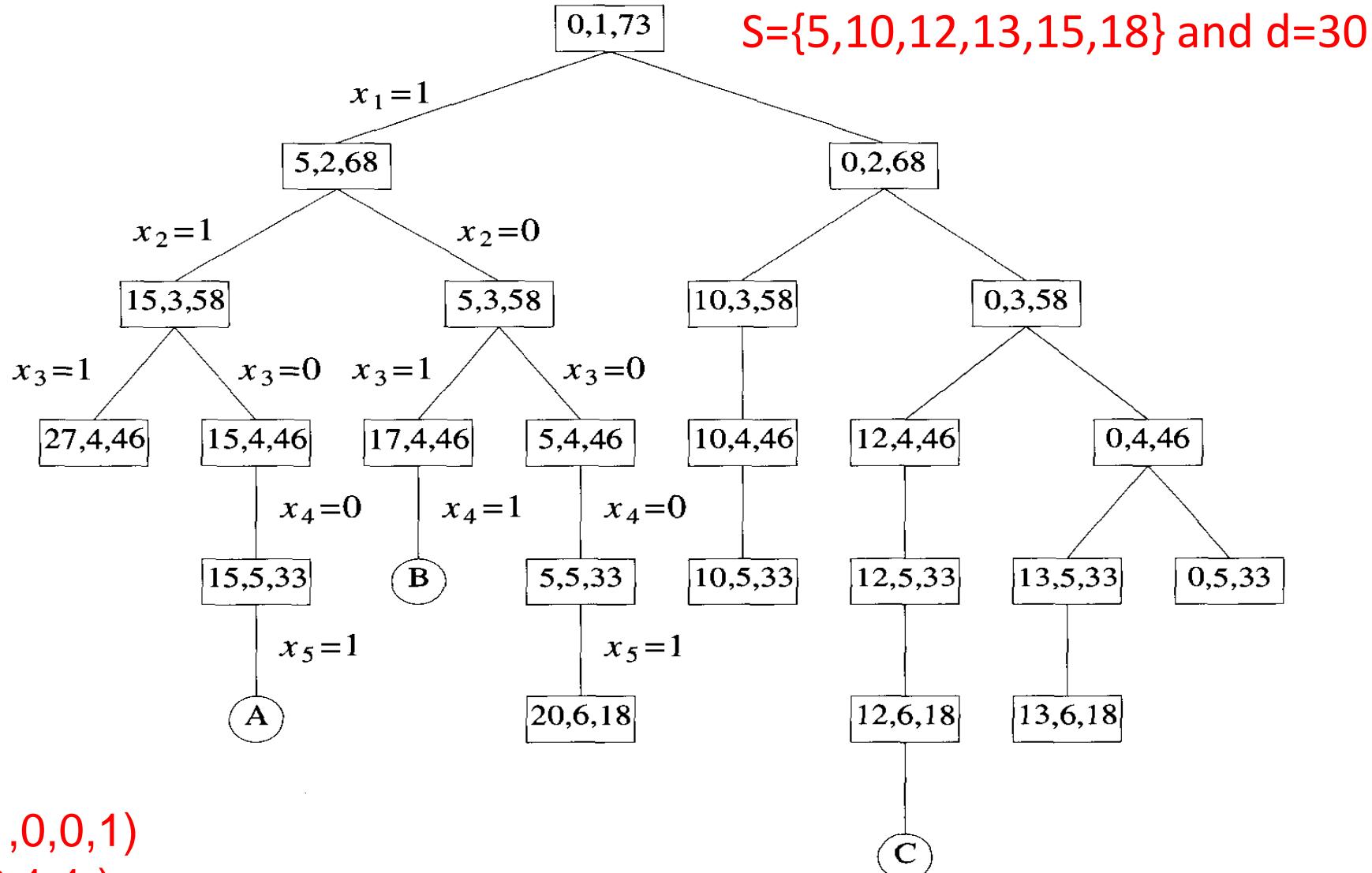
```
1  Algorithm SumOfSub( $s, k, r$ )
2    // Find all subsets of  $w[1 : n]$  that sum to  $m$ . The values of  $x[j]$ ,
3    //  $1 \leq j < k$ , have already been determined.  $s = \sum_{j=1}^{k-1} w[j] * x[j]$ 
4    // and  $r = \sum_{j=k}^n w[j]$ . The  $w[j]$ 's are in nondecreasing order.
5    // It is assumed that  $w[1] \leq m$  and  $\sum_{i=1}^n w[i] \geq m$ .
6  {
7    // Generate left child. Note:  $s + w[k] \leq m$  since  $B_{k-1}$  is true.
8     $x[k] := 1$ ;
9    if ( $s + w[k] = m$ ) then write ( $x[1 : k]$ ); // Subset found
10   // There is no recursive call here as  $w[j] > 0$ ,  $1 \leq j \leq n$ .
11   else if ( $s + w[k] + w[k + 1] \leq m$ )
12     then SumOfSub( $s + w[k], k + 1, r - w[k]$ );
13   // Generate right child and evaluate  $B_k$ .
14   if (( $s + r - w[k] \geq m$ ) and ( $s + w[k + 1] \leq m$ )) then
15   {
16      $x[k] := 0$ ;
17     SumOfSub( $s, k + 1, r - w[k]$ );
18   }
19 }
```

Rules to be followed to construct SOS solution

- If the bounding function is true then a node gets generated otherwise it will not.

$$\sum_{i=1}^k w_i x_i + \sum_{i=k+1}^n w_i \geq m \dots \dots \text{(a) and}$$

Portion of SST for Sum Of Subsets problem



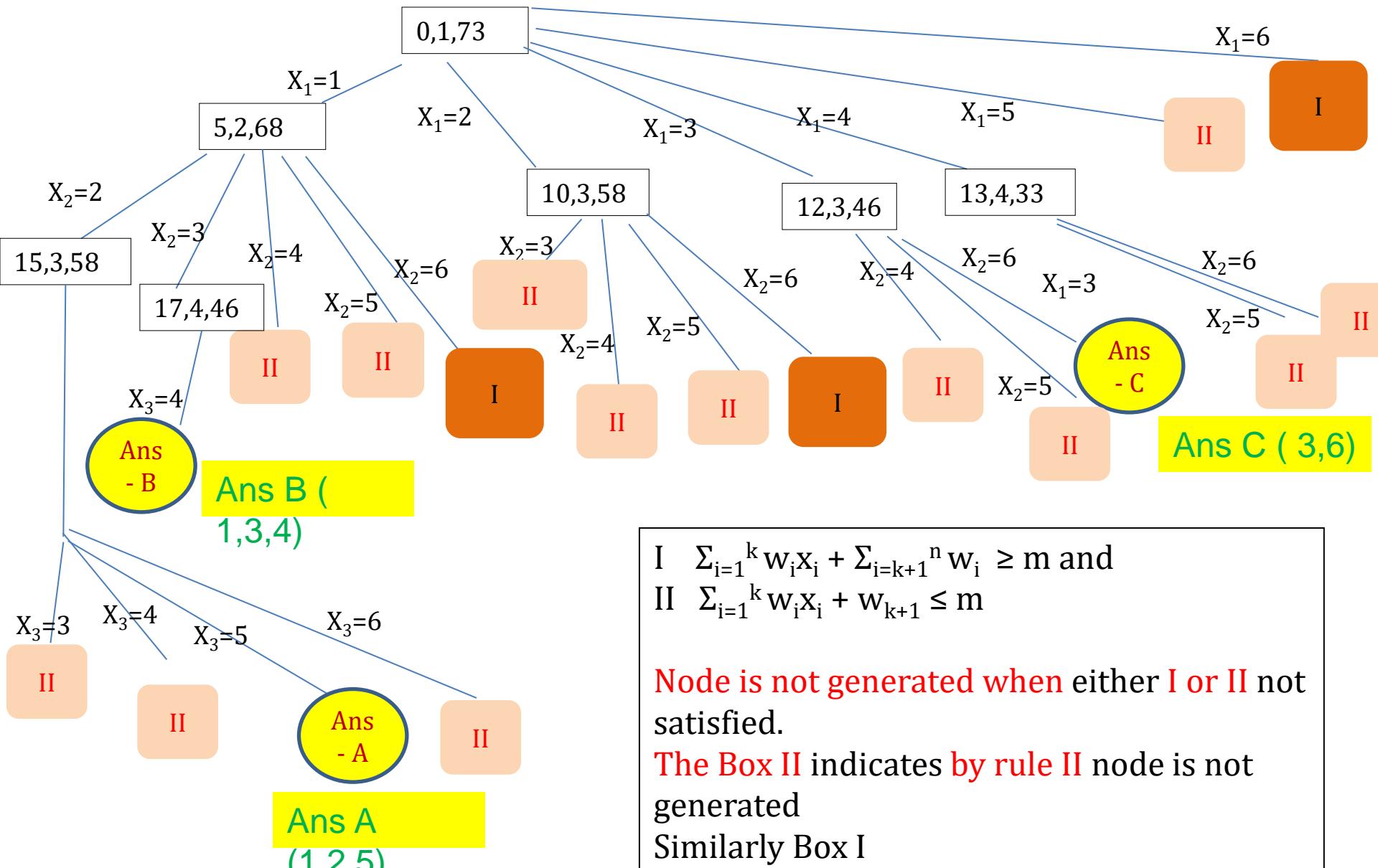
(1,1,0,0,1)

(1,0,1,1,)

(0,0,1,0,0,1) are 3 fixed tuple solutions to the given problem.

Sum of Subsets solution using Variable Tuple Representation

$S=\{5,10,12,13,15,18\}$ and $d=30$



$$\begin{aligned} \text{I } & \sum_{i=1}^k w_i x_i + \sum_{i=k+1}^n w_i \geq m \text{ and} \\ \text{II } & \sum_{i=1}^k w_i x_i + w_{k+1} \leq m \end{aligned}$$

Node is not generated when either I or II not satisfied.

The Box II indicates by rule II node is not generated
Similarly Box I

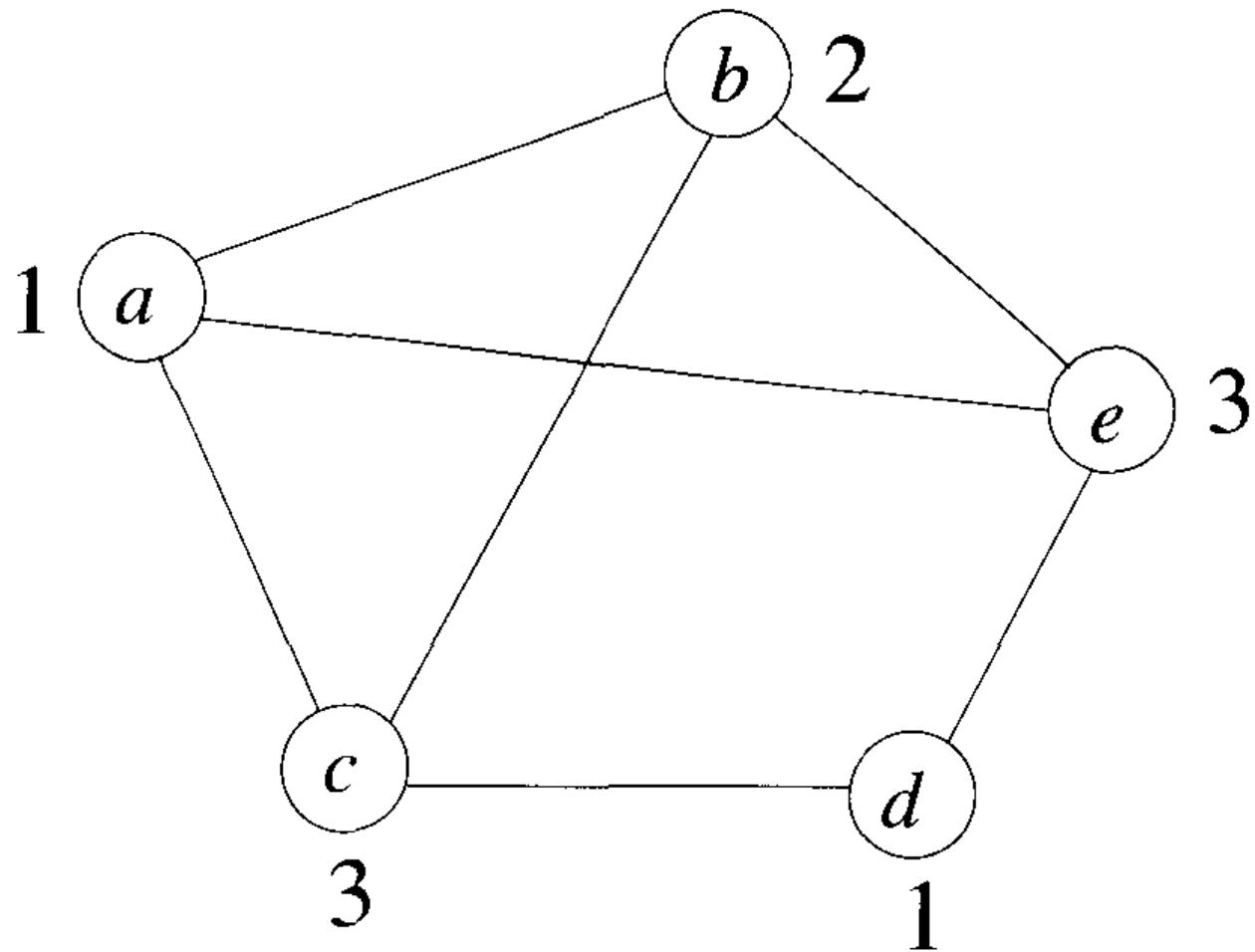
Time Complexity

- The full state space tree for n elements requires $2^n - 1$ nodes from which calls could be made.
- Time complexity of Sum of Subsets is given by $O(2^n)$

Graph coloring Problem

- Let G be a graph and m be a given positive integer. We want to discover whether the nodes of G can be colored in such a way that no two adjacent nodes have the same color yet only m colors are used. This is termed as the m -colorability decision problem.
- If d is the degree of the graph, then graph can be colored with $d+1$ colors.
- m Colorability optimization problem asks for the smallest integer m for which graph G is colored. This integer is referred to as chromatic number.
 - Explicit Constraints : $x_i \in \{j \mid j \text{ is an integer} \& 1 \leq j \leq m\}$
 - Implicit Constraints : No two adjacent x_i 's be same.

Graph Coloring Problem



Planar graph

- A graph is said to be planar iff it can be drawn in a plane in such a way that no two edges cross each other.
- A famous special case of the m-colorability decision problem is the 4-color problem for planar graphs.
- A map can be easily transformed into a graph . Each region of the map becomes a node, and if two regions are adjacent then the corresponding nodes are joined by an edge.
- Any map, requires no more than 4 colors to color the entire map.

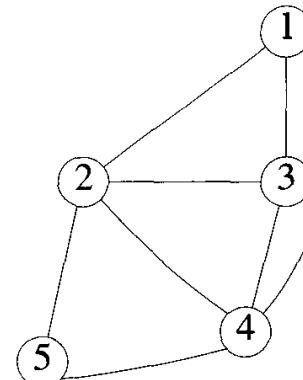
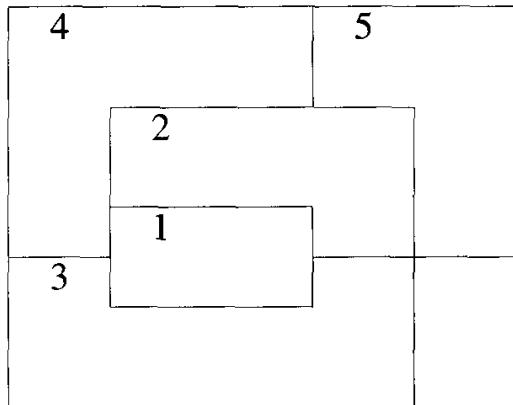
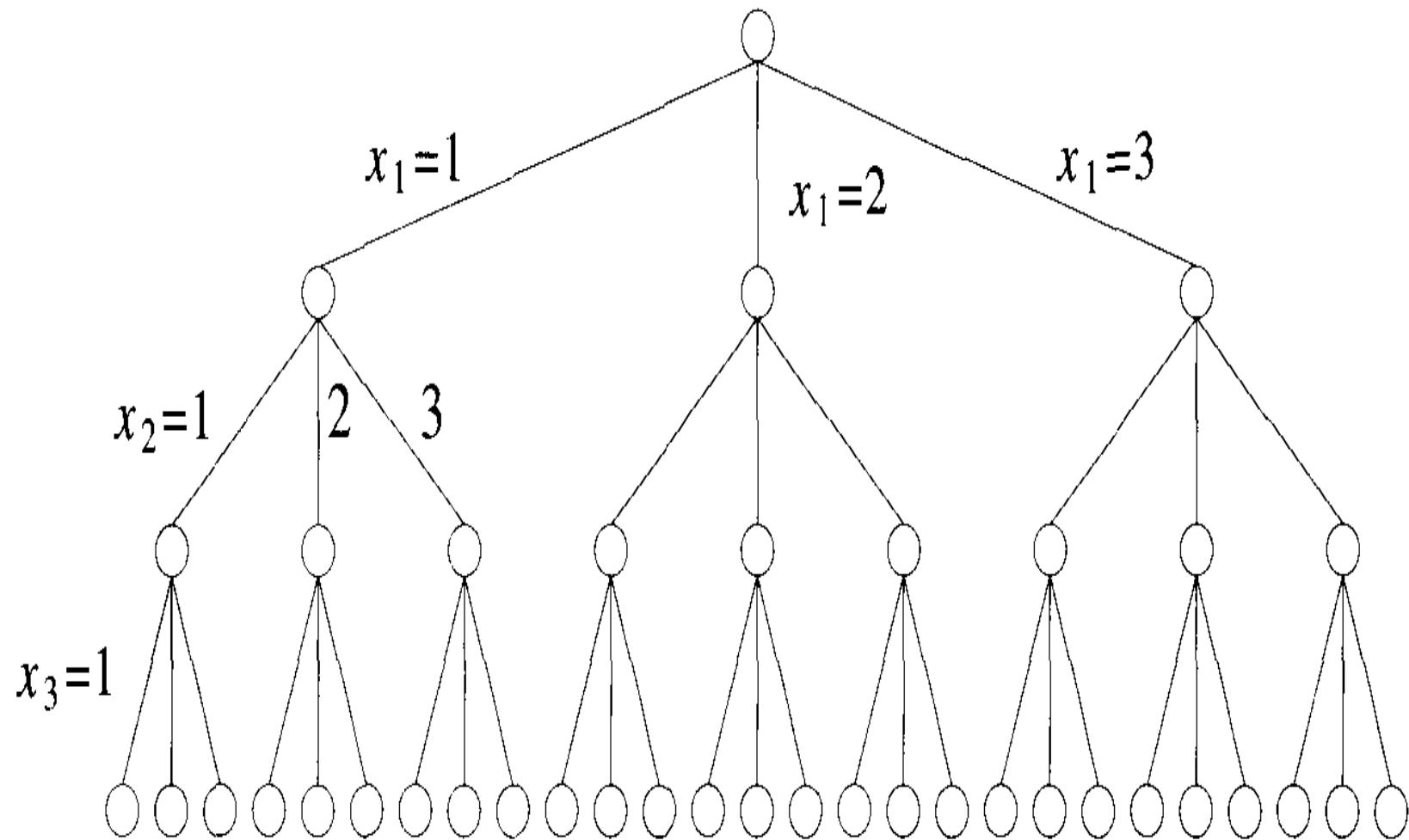


Figure 7.12 A map and its planar graph representation

State Space Tree with $n= 3$, $m = 3$



Graph coloring

```
1 Algorithm mColoring( $k$ )
2 // This algorithm was formed using the recursive backtracking
3 // schema. The graph is represented by its boolean adjacency
4 // matrix  $G[1 : n, 1 : n]$ . All assignments of  $1, 2, \dots, m$  to the
5 // vertices of the graph such that adjacent vertices are
6 // assigned distinct integers are printed.  $k$  is the index
7 // of the next vertex to color.
8 {
9     repeat
10    { // Generate all legal assignments for  $x[k]$ .
11        NextValue( $k$ ); // Assign to  $x[k]$  a legal color.
12        if ( $x[k] = 0$ ) then return; // No new color possible
13        if ( $k = n$ ) then // At most  $m$  colors have been
14                                // used to color the  $n$  vertices.
15            write ( $x[1 : n]$ );
16            else mColoring( $k + 1$ );
17    } until (false);
18 }
```

Graph coloring

```
1  Algorithm NextValue( $k$ )
2    //  $x[1], \dots, x[k - 1]$  have been assigned integer values in
3    // the range  $[1, m]$  such that adjacent vertices have distinct
4    // integers. A value for  $x[k]$  is determined in the range
5    //  $[0, m]$ .  $x[k]$  is assigned the next highest numbered color
6    // while maintaining distinctness from the adjacent vertices
7    // of vertex  $k$ . If no such color exists, then  $x[k]$  is 0.
8  {
9    repeat
10   {
11      $x[k] := (x[k] + 1) \bmod (m + 1)$ ; // Next highest color.
12     if ( $x[k] = 0$ ) then return; // All colors have been used.
13     for  $j := 1$  to  $n$  do
14       { // Check if this color is
15         // distinct from adjacent colors.
16         if (( $G[k, j] \neq 0$ ) and ( $x[k] = x[j]$ ))
17           // If  $(k, j)$  is an edge and if adj.
18           // vertices have the same color.
19           then break;
20       }
21       if ( $j = n + 1$ ) then return; // New color found
22   } until (false); // Otherwise try to find another color.
23 }
```

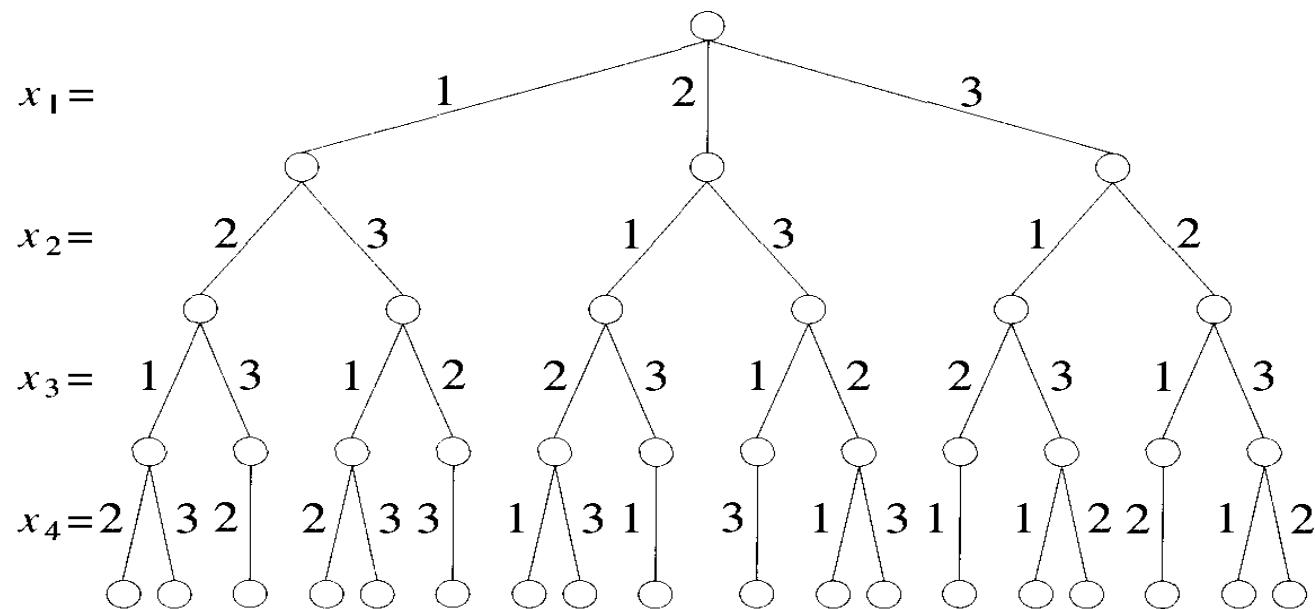


Figure 7.14 A 4-node graph and all possible 3-colorings

Summations

$$\text{i. } \sum_{i=1}^n i = \frac{n(n+1)}{2},$$

$$\text{ii. } \sum_{i=1}^n i^2 = \frac{n(n+1)(2n+1)}{6},$$

$$\text{iii. } \sum_{i=1}^n i^3 = \frac{n^2(n+1)^2}{4},$$

$$\text{iv. } \sum_{i=0}^n r^i = \frac{1-r^{n+1}}{1-r}, \quad r \neq 1 \quad (\text{geometric sum}).$$

Time complexity

- An upper bound on the computing time can be arrived by noticing that the no. of internal nodes in the SST is $\sum_{i=0}^{n-1} m^i$
- At each internal node $O(mn)$ is spent by NextValue to determine next legal color
- The total time is bounded by $\sum_{i=0}^{n-1} m^{i+1}n$ $= \sum_{i=1}^n m^i n$ $= n(m^{n+1} - 2)/(m-1)$ $= O(nm^n)$

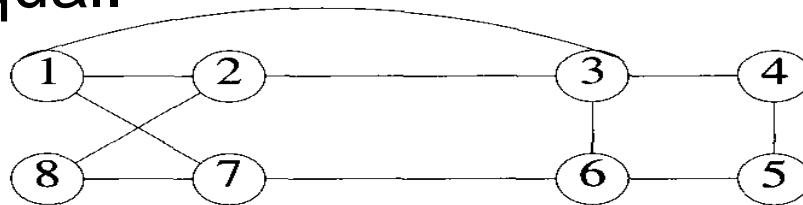
Hamiltonian Cycle

Hamiltonian cycle (HC): is a round trip path along n edges that visits every vertex once and returns to its starting position.

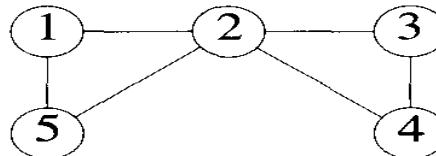
A graph is Hamiltonian iff a Hamiltonian cycle (HC) exists.

If a HC begins at some vertex $v_1 \in G$ and the vertices of G are visited in the order v_1, v_2, \dots, v_{n+1} then the edge (v_i, v_{i+1}) are in G, $1 \leq i \leq n$, and v_i are distinct except for v_1 and v_{n+1} which are equal.

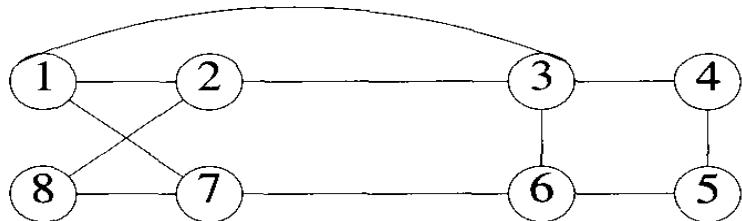
G1:



G2:



G1:

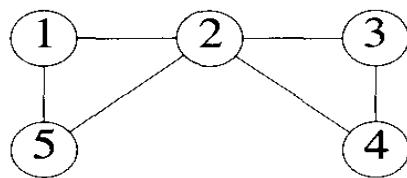


Hamiltonian cycle of
Graph G1

- i. 1-2-8-7-6-5-4-3-1
- ii. 1-3-4-5-6-7-8-2-1
- iii. 2-8-7-6-5-4-3-1-2
- iv. 3-4-5-6-7-8-2-1-3

And so on ...

G2:



No Hamiltonian cycle
possible for Graph G2

Algorithm Hamiltonian

```
1  Algorithm Hamiltonian( $k$ )
2    // This algorithm uses the recursive formulation of
3    // backtracking to find all the Hamiltonian cycles
4    // of a graph. The graph is stored as an adjacency
5    // matrix  $G[1 : n, 1 : n]$ . All cycles begin at node 1.
6    {
7      repeat
8        { // Generate values for  $x[k]$ .
9          NextValue( $k$ ); // Assign a legal next value to  $x[k]$ .
10         if ( $x[k] = 0$ ) then return;
11         if ( $k = n$ ) then write ( $x[1 : n]$ );
12         else Hamiltonian( $k + 1$ );
13       } until ( $\text{false}$ );
14   }
```

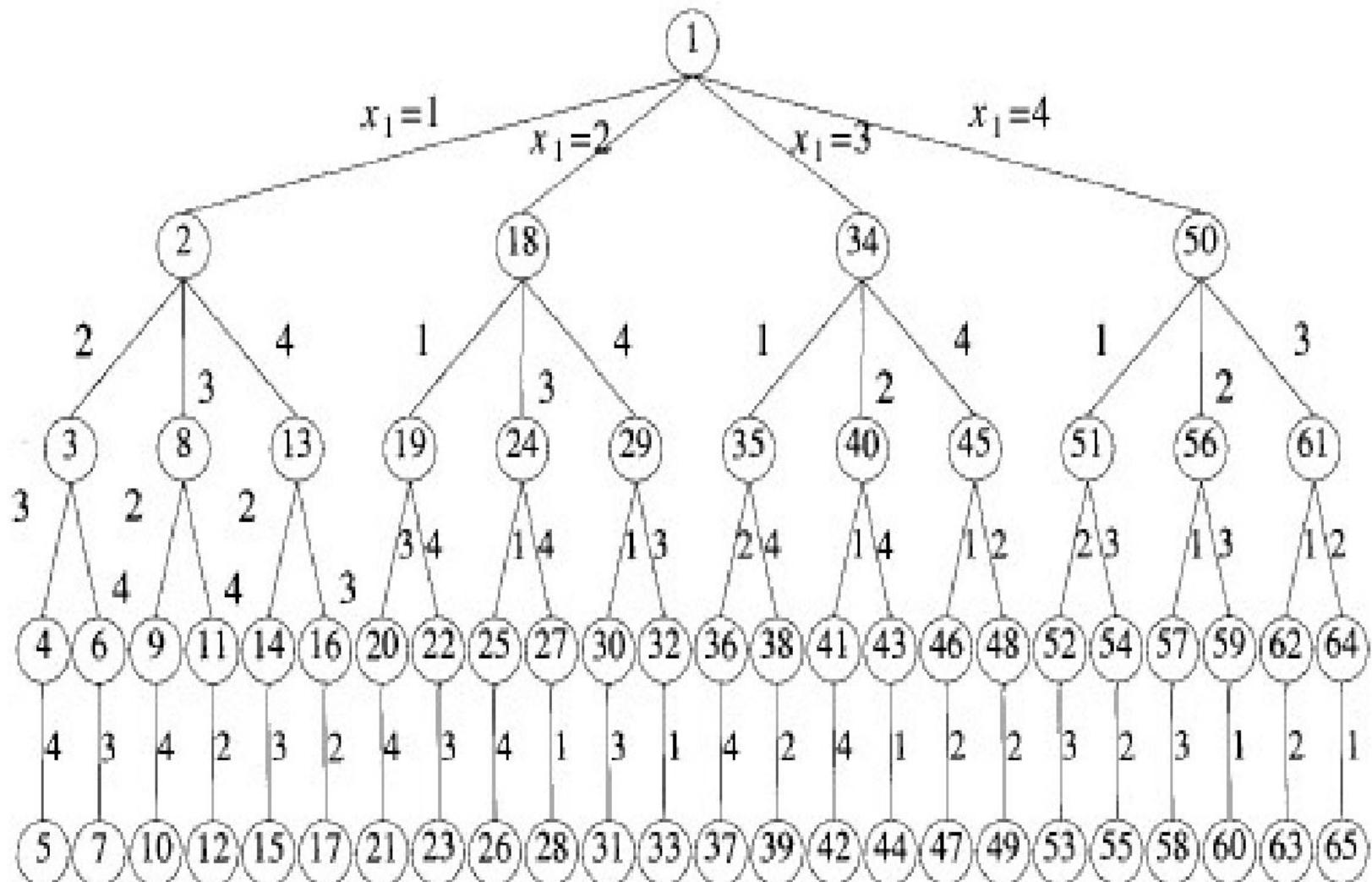
Hamiltonian Cycle Alg

```
1  Algorithm NextValue( $k$ )
2    //  $x[1 : k - 1]$  is a path of  $k - 1$  distinct vertices. If  $x[k] = 0$ , then
3    // no vertex has as yet been assigned to  $x[k]$ . After execution,
4    //  $x[k]$  is assigned to the next highest numbered vertex which
5    // does not already appear in  $x[1 : k - 1]$  and is connected by
6    // an edge to  $x[k - 1]$ . Otherwise  $x[k] = 0$ . If  $k = n$ , then
7    // in addition  $x[k]$  is connected to  $x[1]$ .
8  {
9    repeat
10   {
11      $x[k] := (x[k] + 1) \bmod (n + 1)$ ; // Next vertex.
12     if ( $x[k] = 0$ ) then return;
13     if ( $G[x[k - 1], x[k]] \neq 0$ ) then
14       { // Is there an edge?
15         for  $j := 1$  to  $k - 1$  do if ( $x[j] = x[k]$ ) then break;
16           // Check for distinctness.
17         if ( $j = k$ ) then // If true, then the vertex is distinct.
18           if (( $k < n$ ) or (( $k = n$ ) and  $G[x[n], x[1]] \neq 0$ ))
19             then return;
20       }
21     } until (false);
22 }
```

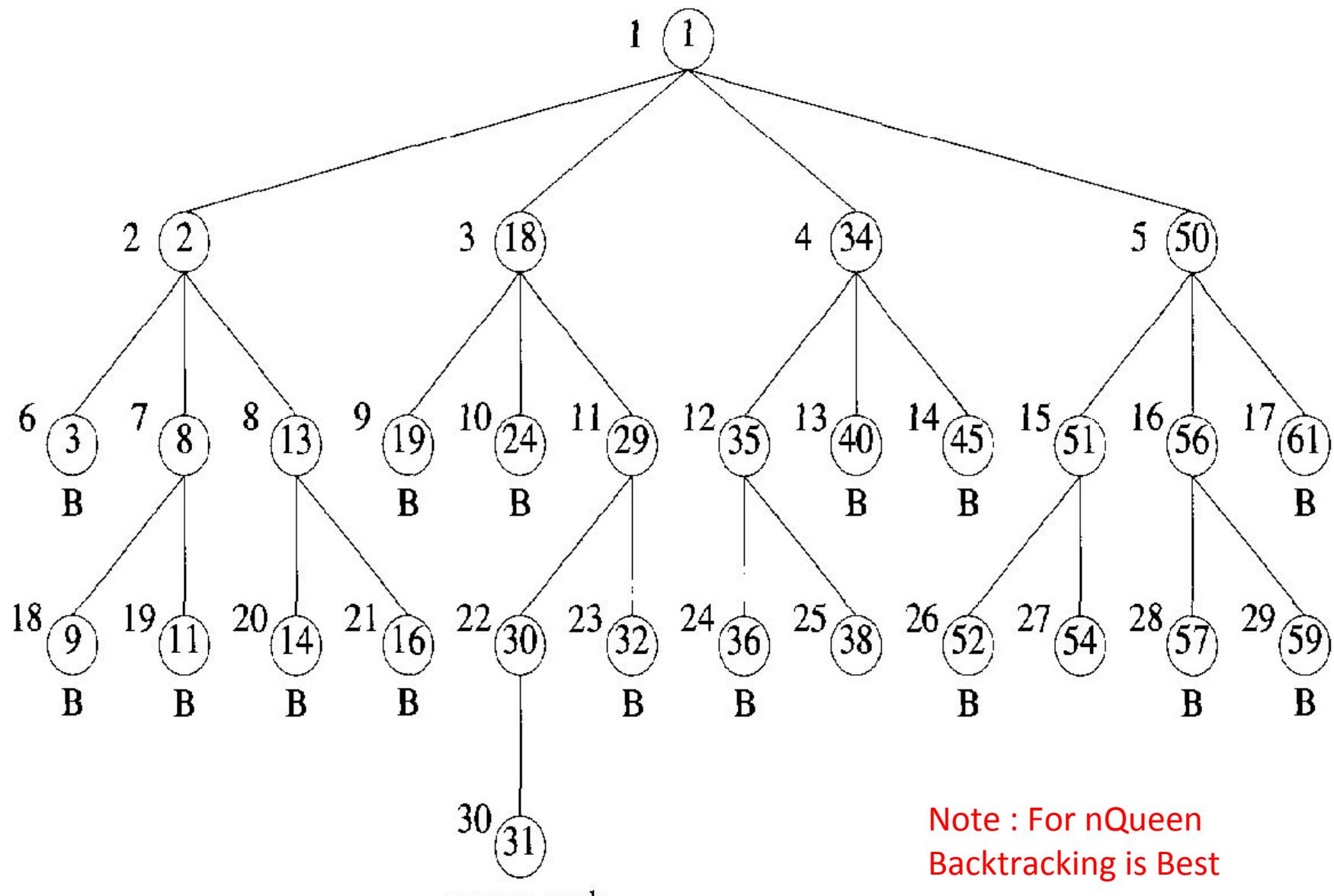
Branch and Bound

- Term Branch and Bound refers to all state space search methods in which **all the children of a E-node are generated before any other live node become the E-node.**
- Branch and Bound Technique is used for optimization problems.
- It is **similar to backtracking** technique but uses **FIFO/LIFO-like search.**
- In BB terminology, a BFS like state space will be called **FIFO search**, the list of Live nodes is a FIFO list or (queue)
- A D-search like state space search will be called **LIFO (last in first out) search**---the list of live nodes is a LIFO list (or stack).
- As in case of backtracking, bounding functions are used to help avoid the generation of subtrees that do not contain an answer node.

FIFO Search of NQueen



Portion of nQueen SSS using FiFo Search



answer node

Portion of nQueen with DFS search

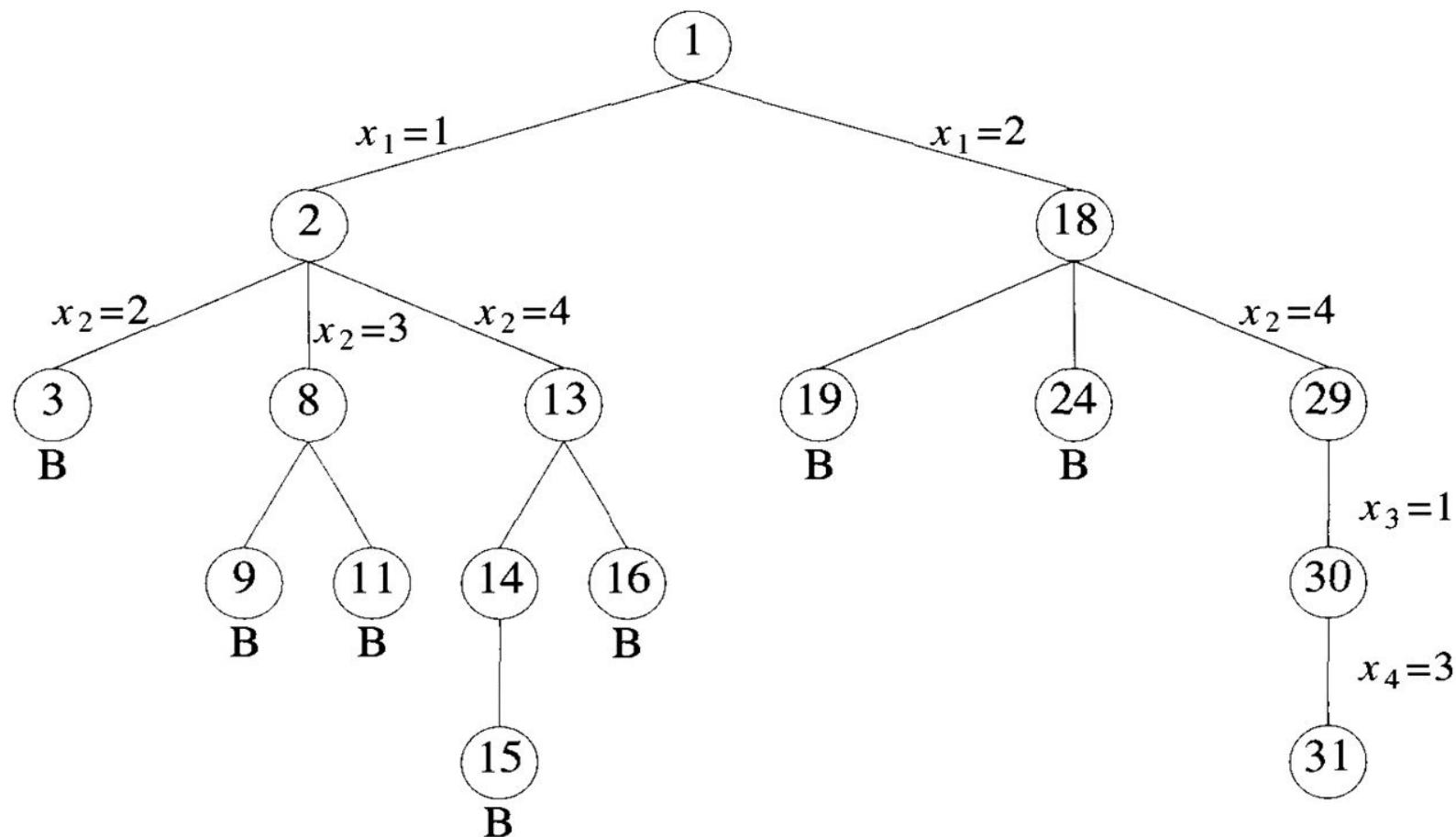


Figure 7.6 Portion of the tree of Figure 7.2 that is generated during backtracking

NQueen

- Nodes that are killed as a result of the bounding functions have a "B" under them.
- A comparison of backtracking & FIFO branch and Bound figures indicates that backtracking is a superior search method for this problem.
- **Least cost (LC) search:**
 - In both LIFO and FIFO branch-and-bound the selection rule for the next E-node is rather rigid.
 - The selection rule for the next E-node does not give any preference to a node that has a very good chance of getting the search to an answer node quickly
 - Thus, in previous example (4-queens)when node 30 is generated, it should have become obvious to the search algorithm that this node will lead to an answer node in one move.
 - However, the rigid FIFO rule first requires the expansion of all live nodes generated before node 30 was expanded.

Least Cost Search

- The search for an answer node can be speeded up by using an “intelligent” ranking function $\hat{c}(.)$ for live nodes.
 - The next E-node is selected on the basis of this ranking function.
- In the 4-queens example if we use a ranking function that assigns node 30 a better rank than all other live nodes, then 30 will become the E-node following node 29.
- The remaining live nodes will never become E-nodes as expansion of node 30 results in the generation of an answer node. (node 31).

Rule to Assign rank

- Assign ranks based on the additional effort needed to reach the answer node from the live nodes.
 - i. The no. of nodes in the subtree that need to be generated before an answer node is generated
 - Always generates minimum no. of nodes
 - ii. The no. of levels the nearest answer node is from x.
 - The only nodes to become E-nodes are the nodes in the path from root to nearest answer node

The difficulty with using either of these ideal cost functions is that computing the cost of a node usually involves a search of the subtree x for an answer node.

- Hence, by the time the cost of a node is determined, that subtree has been searched and there is no need to explore x again.
- For this reason, search algorithms usually rank nodes only on the basis of an estimate $\hat{g}(\cdot)$

LC-Search

- Therefore, the rank of 'x' can be assigned with
 $\hat{c}(x) = f(h(x)) + \hat{g}(x)$
 - i. $h(x)$ is the cost of reaching x from the root and $f(\cdot)$ is any non decreasing function
 - ii. $\hat{g}(x)$ is the estimated additional effort or cost needed to reach an answer node from x
- A search strategy that uses a cost function $\hat{c}(x) = f(h(x)) + \hat{g}(x)$ to select the next E-node would always choose for its next Enode, a live node with least $\hat{c}(\cdot)$. Hence such a strategy is called least cost or LC-Search.
 - $c(\cdot)$ is defined as follows:
 - » if x is an answer node, then $c(x)$ is the cost of reaching x from the root of SST
 - » If x is not an answer node, then $c(x) = \infty$ otherwise
 - » $c(x)$ is the minimum cost answer node in the subtree x .
- The LC-Search expands the node with minimum $\hat{c}(x)$.

LC-Search

- BFS and D-Search are special cases of LC-Search
 - e.g.- if $\hat{g}(x) = 0$ & $f(h(x))=\text{level of node } x$, then LC-Search generates nodes by levels. This is same as BFS.
 - If $f(h(x))=0$ & $\hat{g}(x) \geq \hat{g}(y)$, y is a child of ' x ', then the search is D-Search.
- LC-Search coupled with bounding functions is called as LC Branch and Bound Search.

Bounding

- Assume that each answer node x has a cost $c(x)$ associated with it and that a minimum-cost answer node is to be found.
- A **BB** method searches a **SST** using any search mechanism (**LIFO**, **FIFO**,) in which all the children of the E-node are generated before another node becomes the E-node.
- The cost function $\hat{c}(\cdot)$ such that $\hat{c}(x) \leq c(x)$ is used to provide **lower bounds** on solutions obtainable from a node x .
- Let **upper** be an **upper bound** on the cost of a minimum-cost solution, then all live nodes x with $\hat{c}(x) > \text{upper}$ may be killed as all answer nodes reachable from x have cost $c(x) \geq \hat{c}(x) > \text{upper}$.
 - Initial Value of **upper** can be ∞ or may be obtained by some heuristic measure.
 - So long as the initial value of **upper** $> \hat{c}(x)$, the nodes that can reach the answer node will not get killed.

LC – Search Control Abstraction

```
listnode = record {
    listnode * next, * parent; float cost;
}

1 Algorithm LCSearch(t)
2 // Search t for an answer node.
3 {
4     if *t is an answer node then output *t and return;
5     E := t; // E-node.
6     Initialize the list of live nodes to be empty;
7     repeat
8     {
9         for each child x of E do
10        {
11            if x is an answer node then output the path
12                from x to t and return;
13            Add(x); // x is a new live node.
14            (x → parent) := E; // Pointer for path to root.
15        }
16        if there are no more live nodes then
17        {
18            write ("No answer node"); return;
19        }
20        E := Least();
21    } until (false);
22 }
```

Note : LC, Fifo and Lifo

One should note the similarity between algorithm LCSearch and algorithms for a breadth first search and D -search of a state space tree. If the list of live nodes is implemented as a queue with `Least()` and `Add(x)` being algorithms to delete an element from and add an element to the queue, then LCSearch will be transformed to a FIFO search schema. If the list of live nodes is implemented as a stack with `Least()` and `Add(x)` being algorithms to delete and add elements to the stack, then LCSearch will carry out a LIFO search of the state space tree. Thus, the algorithms for LC, FIFO, and LIFO search are essentially the same. The only difference is in the implementation of the list of live nodes. This is to be expected as the three search methods

0/1 Knapsack Problem

Branch and bound deals with minimization , therefore the objective function is minimized

minimize – $\sum p_i x_i$

$$\text{Minimize} \quad - \sum_{i=1}^n p_i x_i$$

Subject to $\sum_{i=1}^n w_i x_i \leq m$,

$x_i = 0$ or 1 , $1 \leq i \leq m$

0/1 Knapsack Problem

- Every leaf node in the state space tree representing an assignment for which $\sum_{1 \leq i \leq n} w_i x_i \leq m$ is an answer (or solution) node. All other leaf nodes are infeasible.
- For a **minimum-cost** answer node to correspond to any optimal solution, define $c(x) = - \sum_{1 \leq i \leq n} p_i x_i$ for every answer node x .
- The $\text{Cost}(x) = \infty$ for infeasible leaf nodes. For non leaf nodes, $c(x)$ is recursively defined to be $\min\{c(\text{lchild}(x)), c(\text{rchild}(x))\}$
- Two functions $\hat{c}(x)$ and $u(x)$ are needed such that $\hat{c}(x) \leq c(x) \leq u(x)$ for every node x .

LC Branch & Bound 0/1 Knapsack

- The LC branch and bound solution can be obtained using fixed tuple size formulation.
- Steps:
 1. Compute $\hat{c}(.)$ and $u(.)$ (upper bound) for each node
 2. To compute $u(.)$ include the items into knapsack completely till no more items are not accommodatable. The profit obtained forms upper bound(u) with -ve sign (**minimization**) -- (**ie 0/1 knapsack**)
 3. To compute $\hat{c}(.)$ lower bound, take the fraction of the object which was not accommodatable in knapsack with -ve sign. --- (**ie fractional knapsack**)

$$\hat{c} = u + (-(\text{remaining wt of the bag} * \text{pft of next item/ weight}))$$

LC Branch & Bound 0/1 Knapsack

5. If $\hat{c}(x) > u$ then kill node x, otherwise the minimum cost $\hat{c}(x)$ becomes E-node. If \hat{c} is equal for nodes then choose the path by performing difference of \hat{c} & u .
6. Generate children for E-node and repeat the process until all the nodes are covered.
7. The minimum cost $\hat{c}(x)$ becomes the answer node. Once the answer node is found update the ‘u’ value to minimum ‘u’ value of answer node. Trace the path in backward direction from x to root for the solution subset.

0/1 Knapsack Problem : Upper Bound u(x)

```
1 Algorithm UBound( $cp, cw, k, m$ )
2 //  $cp, cw, k$ , and  $m$  have the same meanings as in
3 // Algorithm 7.11.  $w[i]$  and  $p[i]$  are respectively
4 // the weight and profit of the  $i$ th object.
5 {
6      $b := cp; c := cw;$ 
7     for  $i := k + 1$  to  $n$  do
8     {
9         if ( $c + w[i] \leq m$ ) then
10        {
11             $c := c + w[i]; b := b - p[i];$ 
12        }
13    }
14    return  $b;$ 
15 }
```

0/1 Knapsack Problem : Lower Bound $\hat{c}(x)$

```
1  Algorithm Bound( $cp, cw, k$ )
2    //  $cp$  is the current profit total,  $cw$  is the current
3    // weight total;  $k$  is the index of the last removed
4    // item; and  $m$  is the knapsack size.
5    {
6       $b := cp; c := cw;$ 
7      for  $i := k + 1$  to  $n$  do
8        {
9           $c := c + w[i];$ 
10         if ( $c < m$ ) then  $b := b + p[i];$ 
11         else return  $b + (1 - (c - m)/w[i]) * p[i];$ 
12       }
13     return  $b;$ 
14 }
```

$\hat{c}(x)$ and $u(x)$ 0/1 Knapsack BB

Profit	10	10	12	18
Weight	2	4	6	9

M=15

$$\hat{c}(1) = \text{bound}(0,0,0,15) = -38 \quad u(1) = \text{Ubound}(0,0,0,15) = -32$$

$$\hat{c}(2) = \text{bound}(-10,2,1,15) = -38 \quad u(2) = \text{Ubound}(-10,2,1,15) = -32$$

$$\hat{c}(3) = \text{bound}(0,0,1,15) = -32 \quad u(3) = \text{Ubound}(0,0,1,15) = -22$$

$$\hat{c}(4) = \text{bound}(-20,6,2,15) = -38 \quad u(4) = \text{Ubound}(-20,6,2,15) = -32$$

$$\hat{c}(5) = \text{bound}(-10,2,2,15) = -36 \quad u(5) = \text{Ubound}(-10,2,2,15) = -22$$

$\hat{c}(x)$ uses fractional knapsack, the algorithm bound() is used.

$$\hat{c}(1) = (1,1,1,3/9) = 1*-10+1*-10+1*-12+3/9*-18=-10-10-12-6 = -38$$

$$\hat{c}(2) = (1,1,1,3/9) = 1*-10+1*-10+1*-12+3/9*-18=-10-10-12-6 = -38$$

$$\hat{c}(3) = (0,1,1,5/9) = 0*-10+1*-10+1*-12+5/9*-18= 0-10-12-10 = -32$$

Steps to construct LCBB 0/1 Knapsack

1. $u(x)$ is the upper bound of node x .
Computed using **0/1 knapsack procedure**.
2. $\hat{c}(x)$ is the lower bound of node x .
Computed using **fractional Knapsack procedure**.
3. Initially Upper is set to $u(1)$
 - if $u(x) < \text{Upper}$ then update upper with $u(x)$
 - If $\hat{c}(x) > \text{Upper}$ then kill the node x .
4. In BB the E-node should be completely explored before another node becomes E-node.
 - Nodes that get generated must be added to the list of live nodes.
 - The **next E-node** will be the node with **Least estimate cost $\hat{c}(x)$** .
 - If two nodes x_1, x_2 have same $\hat{c}(x_1) = \hat{c}(x_2)$ then select the e-node as the node with minimum of $\{u(x_1) - \hat{c}(x_1), u(x_2) - \hat{c}(x_2)\}$
5. Repeat the steps till one answer state is seen.
 - Nodes that get generated must be added to the list of live nodes

$\hat{c}(x)$ and $u(x)$ 0/1 Knapsack LCBB

Profit	10	10	12	18
Weight	2	4	6	9

Nodes in LC list	1	2	3	4	5
	E	K		LC	E

LC indicates Least Count
E indicates Enode
K indicates Killed

$\hat{c}(x)$ uses fractional knapsack, the algorithm bound() is used.

$$\hat{c}(1) = (1,1,1,3/9) = 1*-10+ 1*-10+1*-12+3/9*-18=-10-10-12-6 = -38$$

$$\hat{c}(2) = (1,1,1,3/9) = 1*-10+ 1*-10+1*-12+3/9*-18=-10-10-12-6 = -38$$

$$\hat{c}(3) = (0,1,1,5/9) = 0*-10+ 1*-10+1*-12+5/9*-18= 0-10-12-10 = -32$$

$u(x)$ uses 0/1 knapsack, the algorithm Ubound() is used for it.

$$u(1) = (1,1,1,0) = 1*-10+ 1*-10+1*-12+0*-18=-10-10-12+0 = -32$$

$$u(2) = (1,1,1,0) = 1*-10+ 1*-10+1*-12+0*-18=-10-10-12+0 = -32$$

$$u(3) = (0,1,1,0) = 0*-10+ 1*-10+1*-12+0*-18= 0-10-12+0 = -22$$

Among $\hat{c}(2)$ and $\hat{c}(3)$, $\hat{c}(2)$ is least. $\hat{c}(2)$ will be next Enode

Note upper = $u(1)$.

$$\hat{c}(4) = (1,1,1,3/9) = 1*-10+ 1*-10+1*-12+3/9*-18=-10-10-12-6 = -38$$

$$u(4) = (1,1,1,0) = 1*-10+ 1*-10+1*-12+0*-18=-10-10-12+0 = -32$$

$$\hat{c}(5) = (1,0,1,7/9) = 1*-10+ 0*-10+1*-12+7/9*-18= -10-0-12-14 = -36$$

$$u(5) = (1,0,1,0) = 1*-10+ 0*-10+1*-12+0*-18= -10-0-12+0 = -22$$

Among $\hat{c}(3)$, $\hat{c}(4)$ and $\hat{c}(5)$, $\hat{c}(4)$ is least. $\hat{c}(4)$ will be next Enode

$\hat{c}(x)$ and $u(x)$ 0/1 Knapsack LCBB

Profit	10	10	12	18
Weight	2	4	6	9

Nodes in LC list	1	2	3	4	5	6	7	8	9
	E	LC		LC			LC		K

$\hat{c}(x)$ uses fractional knapsack, the algorithm bound() is used.

$$\hat{c}(6) = (1,1,1,3/9) = 1 \cdot 10 + 1 \cdot 10 + 1 \cdot 12 + 3/9 \cdot 18 = -10 - 10 - 12 - 6 = -38$$

$$u(6) = (1,1,1,0) = 1 \cdot 10 + 1 \cdot 10 + 1 \cdot 12 + 0 \cdot 18 = -10 - 10 - 12 + 0 = -32$$

$$\hat{c}(7) = (1,1,0,1) = 1 \cdot 10 + 1 \cdot 10 + 0 \cdot 12 + 1 \cdot 18 = -10 - 10 - 0 - 18 = -38$$

$$u(7) = (1,1,0,1) = 1 \cdot 10 + 1 \cdot 10 + 0 \cdot 12 + 1 \cdot 18 = -10 - 10 - 0 - 18 = -38$$

Among $\hat{c}(3), \hat{c}(5), \hat{c}(6)$ and $\hat{c}(7)$, both $\hat{c}(6), \hat{c}(7)$ are least. $\hat{c}(7)$ will be next Enode by taking minimum of $\{u(x_1) - \hat{c}(x_1), u(x_2) - \hat{c}(x_2)\}$

$$\hat{c}(8) = (1,1,0,1) = 1 \cdot 10 + 1 \cdot 10 + 0 \cdot 12 + 1 \cdot 18 = -10 - 10 - 0 - 18 = -38$$

$$u(8) = (1,1,0,1) = 1 \cdot 10 + 1 \cdot 10 + 0 \cdot 12 + 1 \cdot 18 = -10 - 10 - 0 - 18 = -38$$

By Node 8 generation all n x_i 's are assigned either 1/0's. Therefore one solution to the problem is obtained. Now update upper to $\hat{c}(8)$ ie -38

$$\hat{c}(9) = (1,1,0,0) = 1 \cdot 10 + 1 \cdot 10 + 0 \cdot 12 + 0 \cdot 18 = -10 - 10 - 0 - 0 = -20$$

$$u(9) = (1,1,0,0) = 1 \cdot 10 + 1 \cdot 10 + 0 \cdot 12 + 0 \cdot 18 = -10 - 10 - 0 - 0 = -20$$

Node 9 kills as its bounds are > -38 .

LC indicates Least Count

E indicates Enode

K indicates Killed

$\hat{c}(x)$ and $u(x)$ 0/1 Knapsack LCBB

Profit	10	10	12	18	Nodes in LC list	1	2	3	4	5	6	7	8	9
Weight	2	4	6	9		E	LC	K	LC	K	K	LC	ANS	K
						E	E	E	E			E		

By Node 8 generation all x_i 's are assigned either 1/0's. Therefore one solution to the problem is obtained. **At Node 8 we obtained one ANSWER state.**

Now update upper to $\hat{c}(8)$ ie -38

Now check whether there exists any node whose $\hat{c}(x) > \text{upper}$, if so kill such nodes.

Note we have updated Upper = -38

$\hat{c}(3) = -32 > \text{upper}$ node 3 gets killed

$\hat{c}(5) = -36 > \text{upper}$ Node 5 gets killed

$\hat{c}(6) = -38 = \text{upper}$, but its $u(6) = -32 > \text{upper}$ Node 6 gets killed

LC indicates Least Count
E indicates Enode
K indicates Killed
ANS is Answer

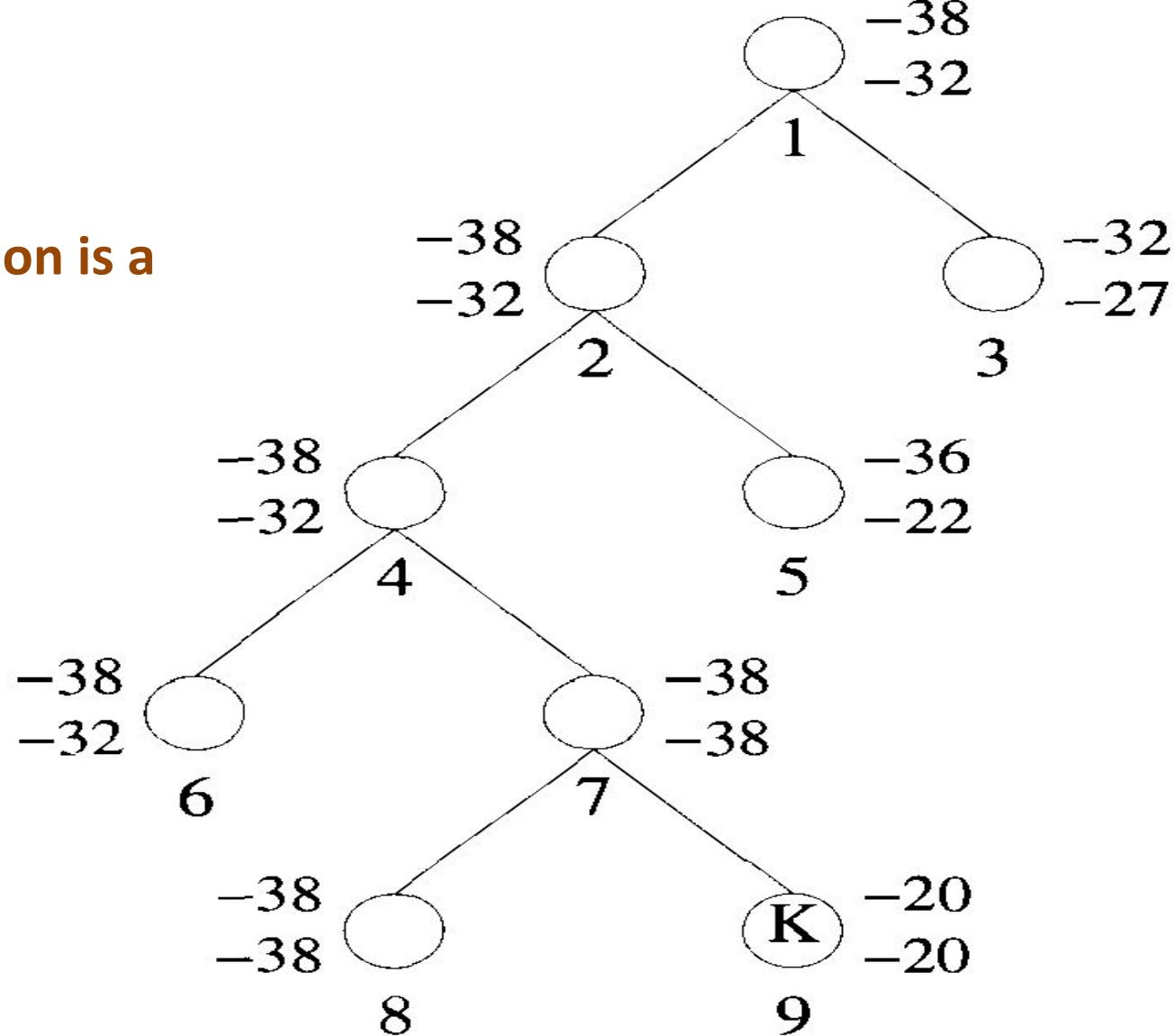
After ascertaining provide solution to knapsack problem. Traverse backwards from node 8 to 7 $x_4=1$, node 7 to 4 $x_3=0$ node 4 to 2 $x_2=1$ and node 2 to 1 $x_1=1$

x_1	x_2	x_3	x_4
1	1	0	1

GRIET/IT/B.Tech III-I/Dr. Y. Sri Lalitha/Unit-IV Branch-Bound

LCBB of 0/1Knapsack

The Tree construction is a must.



Upper number = \hat{c}

Lower number = u

Steps to construct FIFOBB 0/1 Knapsack

1. $u(x)$ is the upper bound of node x .
Computed using **0/1 knapsack procedure**.
2. $\hat{c}(x)$ is the lower bound of node x .
Computed using **fractional Knapsack procedure**.
3. Initially Upper is set to $u(1)$
 - if $u(x) < \text{Upper}$ then update upper with $u(x)$
 - If $\hat{c}(x) > \text{Upper}$ then kill the node x .
4. In BB the E-node should be completely explored before another node becomes E-node.
 - Nodes that get generated must be added to the list of live nodes in FIFO order, therefore Queue is used for it.
 - The **next E-node** is the node at front in queue.
5. Repeat the steps till one answer state is seen.
 - Nodes that get generated must be added to the queue of live nodes

$\hat{c}(x)$ and $u(x)$ 0/1 Knapsack FIFOBB

Profit	10	10	12	18	Nodes in LC list	1	2	3	4	5	6	7	8	9	10	11	12	13
Weight	2	4	6	9		E	E	E	E	K	K	E	E	E	K	K	ANS	K

FIFOBB uses Queue to hold the generated nodes information.

As in Queue it processes those nodes that are first explored.

Initially node 1 is Enode, its upper and lower bounds are computed in the same way as in LCBB. The computation is same as LCBB. The exploration of nodes differ in FIFOBB from that of LCBB

Note upper = $u(1)$.

Node 1 Expansion Generates Nodes 2 and 3

Node 2 becomes Enode. It generates Nodes 4 and 5.

Next Node 3 becomes Enode, since the next node in queue to become Enode is 3.

Generates node 6 and 7. node 7 gets killed as $\hat{c}(7) > \text{upper}$.

Next Enode is node 4, it generates nodes 8 & 9. $u(9) < \text{upper}$, set upper to $u(9)$.

Note : Nodes 5, 6 gets killed by rule $\hat{c}(x) > \text{upper}$.

Next Enode is node 8, it generates Nodes 10 & 11, Node 10 is infeasible as inclusion of 4 items exceeds Knapsack Capacity. Hence killed.

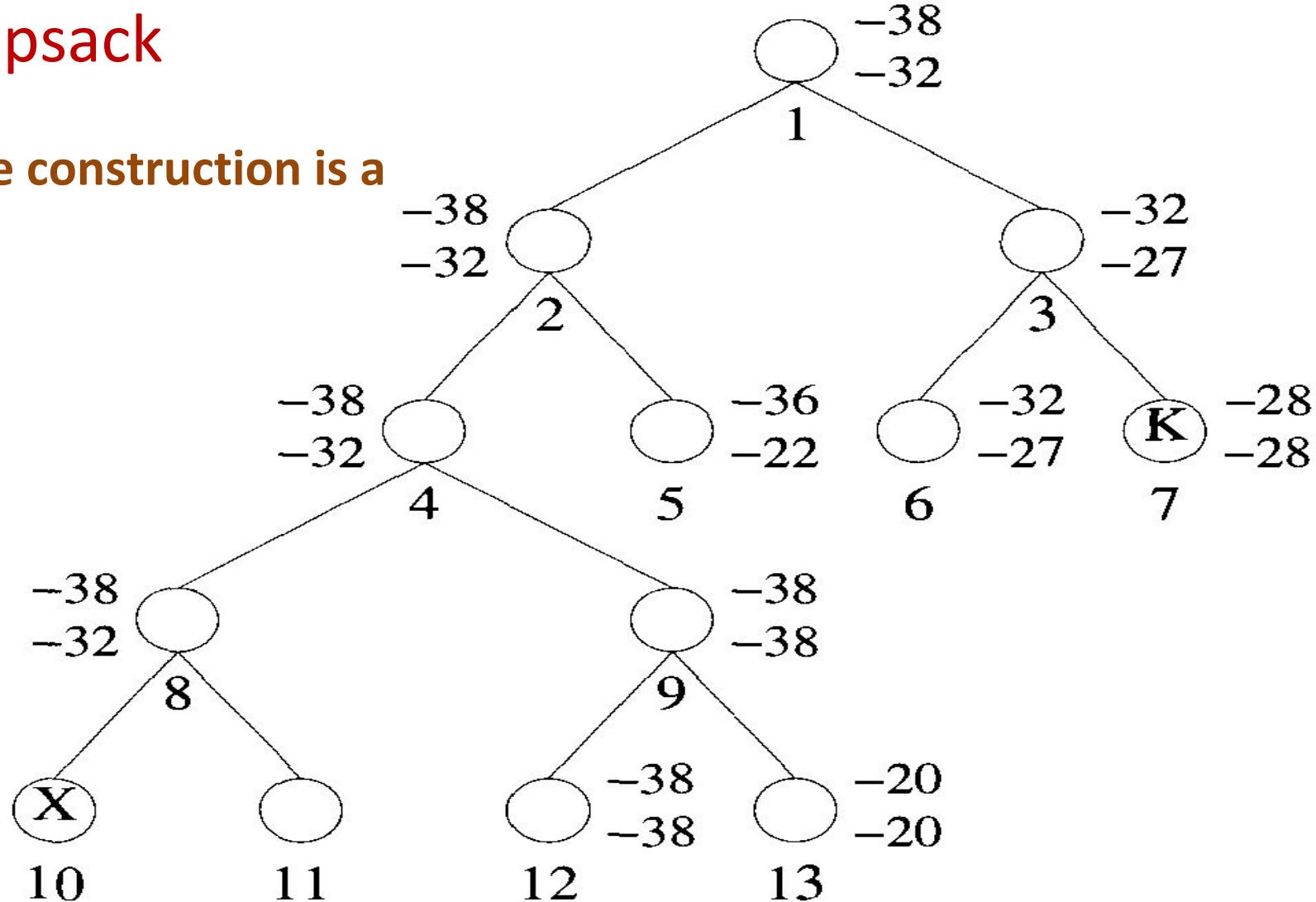
Node 11 gets killed $\hat{c}(11) > \text{upper}$. Similarly Node 13 gets killed

Node 12 is Answer. Like in LCBB construct the solution by traversing from 12 to 1.

x_1	x_2	x_3	x_4
1	1	0	1

FIFOBB of 0/1Knapsack

The Tree construction is a must.



$$\begin{aligned} \text{upper number} &= \hat{c} \\ \text{lower number} &= u \end{aligned}$$

Travelling Sales Person

- **Definition:** Find a tour of minimum cost starting from a node S going through other nodes only once and returning to the starting point S.
- Dynamic TSP takes $O(n^2 2^n)$ time to find a solution.
- BB will not take any less than this time, but for some instance, due to Bounding function, finding an optimal solution is much faster.

Travelling Sales Person

Let $G = (V, E)$ be a directed graph defining an instance of the traveling salesperson problem. Let c_{ij} equal the cost of edge $\langle i, j \rangle$, $c_{ij} = \infty$ if $\langle i, j \rangle \notin E$, and let $|V| = n$. Without loss of generality, we can assume that every tour starts and ends at vertex 1. So, the solution space S is given by $S = \{1, \pi, 1 | \pi \text{ is a permutation of } (2, 3, \dots, n)\}$. Then $|S| = (n - 1)!$. The size of S can be reduced by restricting S so that $(1, i_1, i_2, \dots, i_{n-1}, 1) \in S$ iff $\langle i_j, i_{j+1} \rangle \in E$, $0 \leq j \leq n - 1$, and $i_0 = i_n = 1$. S can be organized into a state space tree

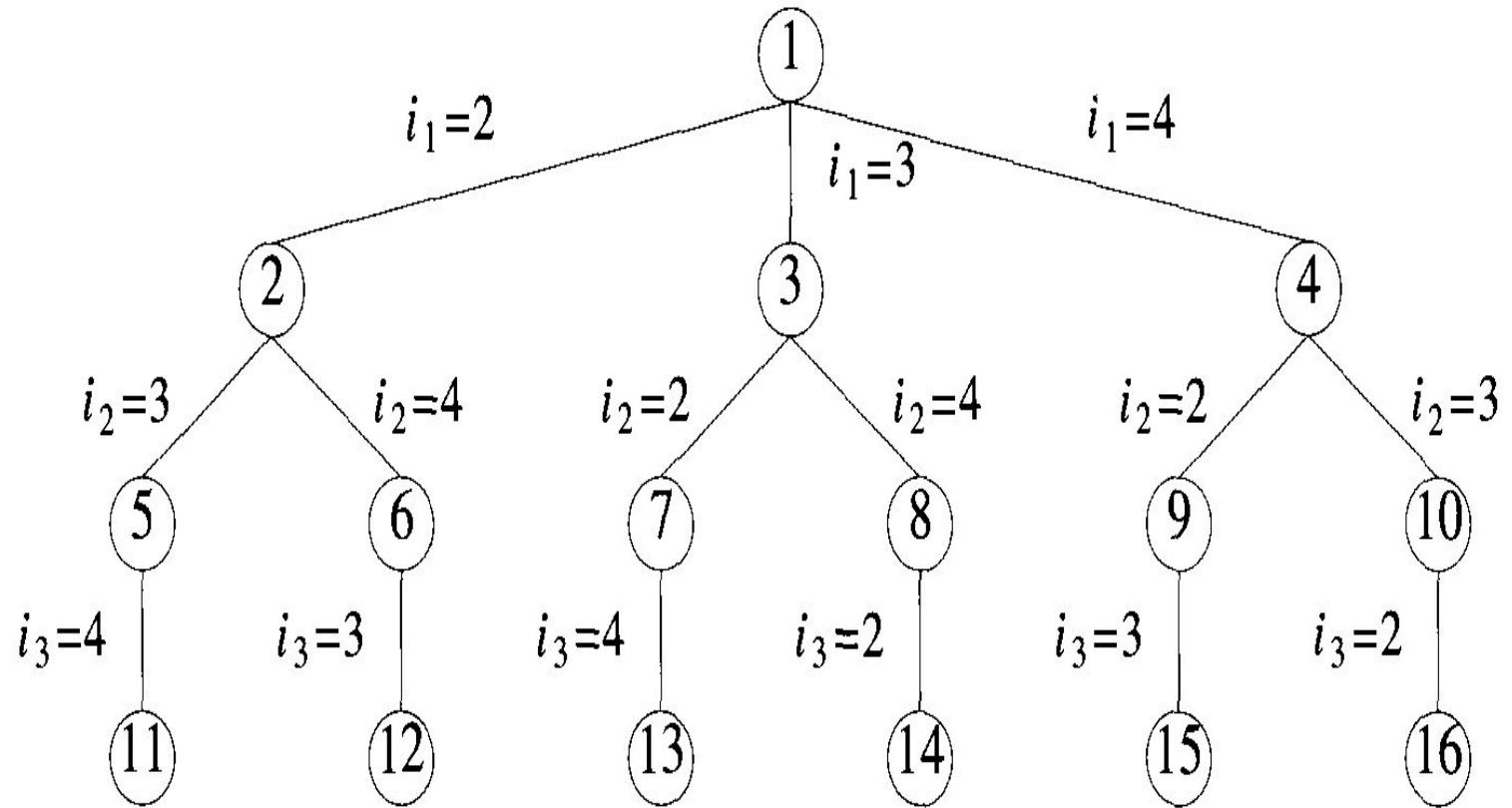


Figure 8.10 State space tree for the traveling salesperson problem with $n = 4$ and $i_0 = i_4 = 1$

Travelling Sales Person

$$c(A) = \begin{cases} \text{length of tour defined by the path from the root to } A, \text{ if } A \text{ is a leaf} \\ \text{cost of a minimum-cost leaf in the subtree } A, \text{ if } A \text{ is not a leaf} \end{cases}$$

Let A be the Reduced Cost Matrix (RCM) for Node R.

Let $\hat{c}(R)$, the lower bound and it is computed as total amount reduced from rows and columns

Let $u(R) = \alpha$

Let S be the child of Node R such that the edge $\langle R, S \rangle$ indicates the inclusion of edge $\langle i, j \rangle$ in the tour.

If S is not leaf node then RCM for S may be obtained as follows

- Set all entries in Row i and Column j of A to α .
 - Prevents the use of any more edges leaving vertex i entering j
- Set $A(j, 1)$ to α .
 - Prevents the use of edge $\langle j, 1 \rangle$
- Reduce S and let r be the reduced cost.

$$\hat{c}(S) = \hat{c}(R) + r + A(i, j)$$

TSP Bounding Conditions

- Initially we assume u (upper bound) = ∞ , as soon as we get one solution (leaf node) we update ‘ u ’ to the value of \hat{c} of the solution (leaf) node .
- Then we kill all those nodes whose $\hat{c}(x) \geq u$.
- Exploration of the unbounded nodes continue till we finish all the nodes.
- The final value in u represents the value of the solution.

TSP Problem

∞	20	30	10	11
15	∞	16	4	2
3	5	∞	2	4
19	6	18	∞	3
16	4	7	16	∞

(a) Cost matrix

∞	10	17	0	1
12	∞	11	2	0
0	3	∞	0	2
15	3	12	∞	0
11	0	0	12	∞

(b) Reduced cost matrix
 $L = 25$

Find the reduced matrix for Cost Matrix (a)

Reduced matrix is the one with atleast one zero in every row and every column.

To make this happen subtract min row value from all the row elements.
 Similarly for column.

R1 - 10, R2 - 2, R3 - 2, R4 - 3, R5 - 4, c1 - 1, c3 - 3

$L = 10+2+2+3+4+1+3 = 25$. Ensure the tour cost cannot be < 25

$\hat{c}(1) = 25$, upper = ∞

The tour begins at vertex 1, in SST node 1 represent Initial state.
 Node 1 is Endode.

LCBB TSP

Nodes in LC list	1	2	3	4	5
	E	L	L	L	L

Node 2 Computation : $i_0 = 1, i_1 = 2.$

edge 1-2 is considered. Let i,j indicate the vertices of edge.

Cost of edge $\langle 1,2 \rangle$ is: $A(1,2) = 10$

- Set **row 1 = ∞** since we are choosing edge $\langle 1,2 \rangle$
- Set **column 2 = ∞** since we are choosing edge $\langle 1,2 \rangle$
- Set **$A(2,1) = \infty$**
- The **resulting cost matrix** is:
 - Which is already in reduced matrix, hence $r=0$.

$$\begin{bmatrix} \infty & \infty & \infty & \infty & \infty \\ \infty & \infty & 11 & 2 & 0 \\ 0 & \infty & \infty & 0 & 2 \\ 15 & \infty & 12 & \infty & 0 \\ 11 & \infty & 0 & 12 & \infty \end{bmatrix}$$

(a) Path 1,2; node 2

The **cost of node 2** (Considering vertex 2 from vertex 1)
 $\hat{c}(2) = \hat{c}(1) + A(1,2) + r = 25 + 10 + 0 = 35$

LCBB TSP

Nodes in LC list	1	2	3	4	5
	E			LC E	

Similarly

Node 3 Computation : $i_0 = 1, i_1 = 3.$

$$\hat{c}(3) = \hat{c}(1) + A(1,3) + r = 25 + 17 + 11 = 53$$

The resulting cost matrix is:

∞	∞	∞	∞	∞
1	∞	∞	2	0
∞	3	∞	0	2
4	3	∞	∞	0
0	0	∞	12	∞

(b) Path 1,3; node 3

Node 4 Computation : $i_0 = 1, i_1 = 4.$

$$\hat{c}(4) = \hat{c}(1) + A(1,4) + r = 25 + 0 + 0 = 25$$

The resulting cost matrix is:

∞	∞	∞	∞	∞
12	∞	11	∞	0
0	3	∞	∞	2
∞	3	12	∞	0
11	0	0	∞	∞

(c) Path 1,4; node 4

Node 5 Computation : $i_0 = 1, i_1 = 5.$

$$\hat{c}(5) = \hat{c}(1) + A(1,5) + r = 25 + 1 + 5 = 31$$

The resulting cost matrix is:

∞	∞	∞	∞	∞
10	∞	9	0	∞
0	3	∞	0	∞
12	0	9	∞	∞
∞	0	0	12	∞

(d) Path 1,5; node 5

LCBB TSP

Nodes in LC list	1	2	3	4	5	6	7	8	
	E			€	LC				

Node 4 becomes next Enode, consider matrix C as A.

Edge 1-4 is selected already.

At vertex 6 edge 4-2 ie : $i_1 = 4, i_2 = 2$ is considered.

$$\hat{c}(6) = \hat{c}(4) + A(4,2) + r = 25 + 3 + 0 = 28$$

The resulting cost matrix is:

At vertex 7 edge 4-3 ie : $i_1 = 4, i_2 = 3$ is considered.

$$\hat{c}(7) = \hat{c}(4) + A(4,3) + r = 25 + 12 + 13 = 50$$

The resulting cost matrix is:

At vertex 8 edge 4-5 ie : $i_1 = 4, i_2 = 5$ is considered.

$$\hat{c}(8) = \hat{c}(4) + A(4,5) + r = 25 + 0 + 11 = 36$$

The resulting cost matrix is:

Next Enode is node 6

∞	∞	∞	∞	∞
∞	∞	11	∞	0
0	∞	∞	∞	2
∞	∞	∞	∞	∞
11	∞	0	∞	∞

(e) Path 1,4,2; node 6

∞	∞	∞	∞	∞
1	∞	∞	∞	0
∞	1	∞	∞	0
∞	∞	∞	∞	∞
0	0	∞	∞	∞

(f) Path 1,4,3; node 7

∞	∞	∞	∞	∞
1	∞	0	∞	∞
0	3	∞	∞	∞
∞	∞	∞	∞	∞
∞	0	0	∞	∞

(g) Path 1,4,5; node 8

LCBB TSP

Nodes in LC list	1	2	3	4	5	6	7	8	9	10	11
	E			€ E		LC E			LC E		

Node 6 becomes next Enode, consider matrix e as A.

Edge 1-4-2 are selected already.

At vertex 9 edge 2-3 ie : $i_2 = 2, i_3 = 3$ is considered.

$$\hat{c}(9) = \hat{c}(6) + A(2,3) + r = 28 + 11 + 13 = 52$$

The resulting cost matrix is:

At vertex 10 edge 2-5 ie : $i_2 = 2, i_3 = 5$ is considered.

$$\hat{c}(10) = \hat{c}(6) + A(2,5) + r = 28 + 0 + 0 = 28$$

The resulting cost matrix is:

At vertex 11 edge 5-3 ie : $i_3 = 5, i_4 = 3$ is considered.

$$\hat{c}(11) = \hat{c}(10) + A(5,3) + r = 28 + 0 + 0 = 28$$

The resulting cost matrix is all infinities

Path is 1-4-2-5-3-1

∞	∞	∞	∞	∞
∞	∞	∞	∞	∞
∞	∞	∞	∞	0
∞	∞	∞	∞	∞
0	∞	∞	∞	∞

(h) Path 1,4,2,3; node 9

∞	∞	∞	∞	∞
∞	∞	∞	∞	∞
0	∞	∞	∞	∞
∞	∞	∞	∞	∞
∞	∞	0	∞	∞

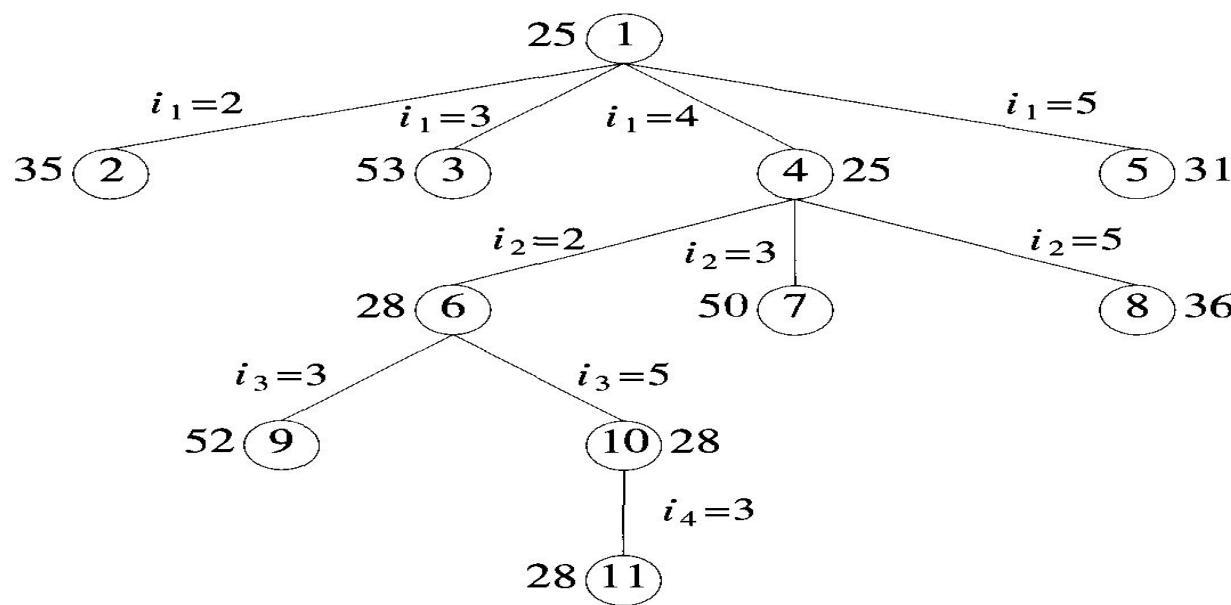
(i) Path 1,4,2,5; node 10

∞	20	30	10	11
15	∞	16	4	2
3	5	∞	2	4
19	6	18	∞	3
16	4	7	16	∞

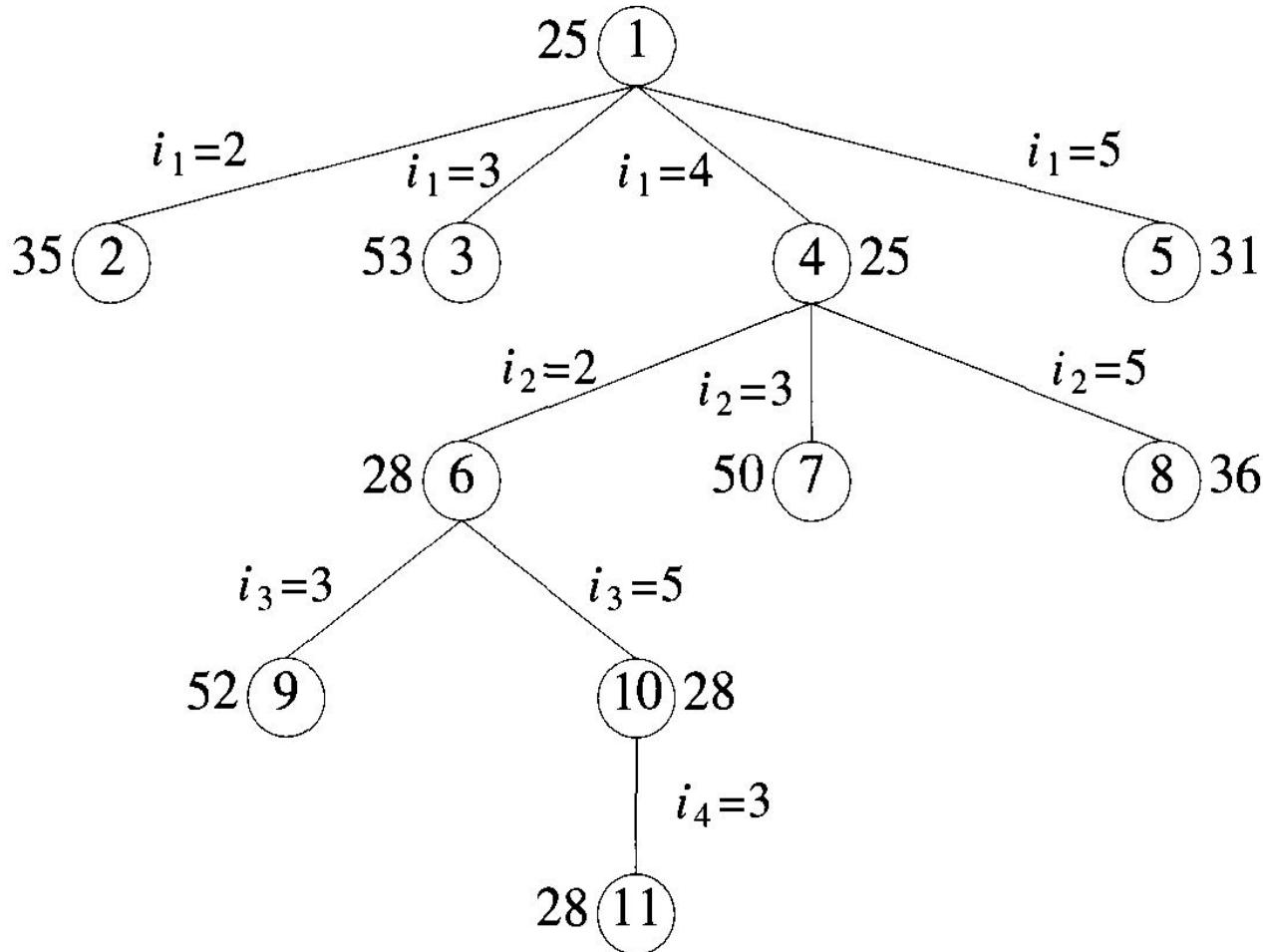
(a) Cost matrix

∞	10	17	0	1
12	∞	11	2	0
0	3	∞	0	2
15	3	12	∞	0
11	0	0	12	∞

(b) Reduced cost
matrix
 $L = 25$



Numbers outside the node are \hat{c} values



Numbers outside the node are \hat{c} values

Figure 8.12 State space tree generated by procedure LCBB

$$\begin{bmatrix} \infty & \infty & \infty & \infty & \infty \\ \infty & \infty & 11 & 2 & 0 \\ 0 & \infty & \infty & 0 & 2 \\ 15 & \infty & 12 & \infty & 0 \\ 11 & \infty & 0 & 12 & \infty \end{bmatrix}$$

(a) Path 1,2; node 2

$$\begin{bmatrix} \infty & \infty & \infty & \infty & \infty \\ 1 & \infty & \infty & 2 & 0 \\ \infty & 3 & \infty & 0 & 2 \\ 4 & 3 & \infty & \infty & 0 \\ 0 & 0 & \infty & 12 & \infty \end{bmatrix}$$

(b) Path 1,3; node 3

$$\begin{bmatrix} \infty & \infty & \infty & \infty & \infty \\ 12 & \infty & 11 & \infty & 0 \\ 0 & 3 & \infty & \infty & 2 \\ \infty & 3 & 12 & \infty & 0 \\ 11 & 0 & 0 & \infty & \infty \end{bmatrix}$$

(c) Path 1,4; node 4

$$\begin{bmatrix} \infty & \infty & \infty & \infty & \infty \\ 10 & \infty & 9 & 0 & \infty \\ 0 & 3 & \infty & 0 & \infty \\ 12 & 0 & 9 & \infty & \infty \\ \infty & 0 & 0 & 12 & \infty \end{bmatrix}$$

(d) Path 1,5; node 5

$$\begin{bmatrix} \infty & \infty & \infty & \infty & \infty \\ \infty & \infty & 11 & \infty & 0 \\ 0 & \infty & \infty & \infty & 2 \\ \infty & \infty & \infty & \infty & \infty \\ 11 & \infty & 0 & \infty & \infty \end{bmatrix}$$

(e) Path 1,4,2; node 6

$$\begin{bmatrix} \infty & \infty & \infty & \infty & \infty \\ 1 & \infty & \infty & \infty & 0 \\ \infty & 1 & \infty & \infty & 0 \\ \infty & \infty & \infty & \infty & \infty \\ 0 & 0 & \infty & \infty & \infty \end{bmatrix}$$

(f) Path 1,4,3; node 7

$$\begin{bmatrix} \infty & \infty & \infty & \infty & \infty \\ 1 & \infty & 0 & \infty & \infty \\ 0 & 3 & \infty & \infty & \infty \\ \infty & \infty & \infty & \infty & \infty \\ \infty & 0 & 0 & \infty & \infty \end{bmatrix}$$

(g) Path 1,4,5; node 8

$$\begin{bmatrix} \infty & \infty & \infty & \infty & \infty \\ \infty & \infty & \infty & \infty & \infty \\ \infty & \infty & \infty & \infty & 0 \\ \infty & \infty & \infty & \infty & \infty \\ 0 & \infty & \infty & \infty & \infty \end{bmatrix}$$

(h) Path 1,4,2,3; node 9

$$\begin{bmatrix} \infty & \infty & \infty & \infty & \infty \\ \infty & \infty & \infty & \infty & \infty \\ 0 & \infty & \infty & \infty & \infty \\ \infty & \infty & \infty & \infty & \infty \\ \infty & \infty & 0 & \infty & \infty \end{bmatrix}$$

(i) Path 1,4,2,5; node 10

Figure 8.13 Reduced cost matrices corresponding to nodes in Figure 8.12

NP Hard - Np Complete problems

Basic Concepts

Basic Concepts

- This chapter is concerned with the distinction between problems that can be solved in polynomial time and problems for which no polynomial time algorithms exists .
- The computing times of algorithms fall into two Categories.
 - Category-1 : Contains problems that can be solved by polynomial time algorithm.
 - Ex : Binary search- $O(\log n)$, sorting- $O(n \log n)$, matrix multiplication $O(n^3)$.
 - Category 2 : Contains problems for which no polynomial time algorithm is known
 - ie., problems whose best known algorithms are non-polynomial.
 - Ex : –Traveling salesperson problem $O(n^2 2^n)$, knapsack problem $O(2^{n/2})$, Sum of subsets $O(2^n)$, Graph coloring $O(n m^n)$ etc.

Deterministic - Non-deterministic

- **Deterministic algorithms**
 - Algorithms in which the outcome (result) of every operation is uniquely defined (ex : Sorting, Searching,)
 - Predictable in terms of output for a certain input, ie for a given input the system gives same output.
 - Also called as **tractable problems**

Deterministic - Non-deterministic

- Nondeterministic algorithms
 - Algorithms in which the outcomes of operations are not uniquely defined but are limited to specific sets of possibilities.
 - Also called as Intractable Problems
 - For the same input, the system produces different output in different runs.
 - $X = \text{choice}(1:10)$ ↗ x can take any integer 1:10
 - These algorithms cannot solve given problem in polynomial time.
 - The machine capable of executing such operations is allowed to choose any one of these outcomes subject to a termination condition.
 - To specify such operations, 3 functions Choice(), Failure() and Success() are introduced with $O(1)$ time complexity each.

Non-deterministic algorithms

1. choice (S) : Arbitrarily chooses one of the elements of set S. The assignment statement $x = \text{choice}(1,n)$ can result in x being assigned any of the integers in the range [1: n], *in a completely arbitrary manner*, there is No rule to specify how this choice is to be made.
2. failure() : Signals an unsuccessful completion. A nondeterministic algorithm terminates unsuccessfully if and only if there exist no set of choices leading to a success signal.
3. success() : Signals a successful completion. whenever there is a set of choices that leads to a successful completion, then one such set of choices is *always* be made and the algorithm terminates successfully
 - The computing time of choice(), success(), failure() are taken to be O(1).
 - A machine capable of executing a nondeterministic algorithm as above is called a **nondeterministic** machine

```
1   $j := \text{Choice}(1, n);$ 
2   $\text{if } A[j] = x \text{ then } \{\text{write }(j); \text{Success}();\}$ 
3   $\text{write }(0); \text{Failure}();$ 
```

gorithm 11.1 Nondeterministic search

```
1  Algorithm NSort( $A$ ,  $n$ )
2    // Sort  $n$  positive integers.
3    {
4      for  $i := 1$  to  $n$  do  $B[i] := 0$ ; // Initialize  $B[ ]$ .
5      for  $i := 1$  to  $n$  do
6        {
7           $j := \text{Choice}(1, n);$ 
8          if  $B[j] \neq 0$  then Failure();
9           $B[j] := A[i];$ 
10         }
11        for  $i := 1$  to  $n - 1$  do // Verify order.
12          if  $B[i] > B[i + 1]$  then Failure();
13        write ( $B[1 : n]$ );
14        Success();
15    }
```

Algorithm 11.2 Nondeterministic sorting

```
1  Algorithm DKP( $p, w, n, m, r, x$ )
2  {
3       $W := 0; P := 0;$ 
4      for  $i := 1$  to  $n$  do
5      {
6           $x[i] := \text{Choice}(0, 1);$ 
7           $W := W + x[i] * w[i]; P := P + x[i] * p[i];$ 
8      }
9      if  $((W > m) \text{ or } (P < r))$  then Failure();
10     else Success();
11 }
```

Algorithm 11.4 Nondeterministic knapsack algorithm

Decision problems vs Optimization problems

- Most problems that do not yield polynomial-time algorithms are either **optimization or decision problems**.
- A class of problem for which the solution is either **zero or one** is called a **decision problem**.
 - Ex : Does an element is present in array?
 - Ex : Does the graph chromatic number is 3
 - *An algorithm for a decision problem is termed a decision algorithm.*
- Any problem that finds an optimal (either min. or max.) value for a given problem is known as an **optimization problem**.
 - Ex : TSP, MST, Knapsack etc
 - *An optimization algorithm is used to solve an optimization problem*

Examples of Decision/Optimization

- Many problems will have decision and optimization versions
 - Eg: Traveling salesman problem
 - **Optimization:** find Hamiltonian cycle of minimum weight
 - **Decision:** is there a Hamiltonian cycle of weight
 - Eg: 0/1 Knapsack
 - **Optimization:** find maximum profit subject to Capacity of Bag
 - **Decision:** is there a solution with profit > r subject to Capacity of Bag

Classes of Problems – P Class

- **P Class** : The class of problems that are solvable in polynomial-time with deterministic algorithms. Also called **Tractable Problems**
 - More specifically, they are problems that can be solved in time $O(n^k)$ for some constant k , where n is the size of the input to the problem
 - Sorting $O(n\log n)$, Searching $O(n)$, Fractional Knapsack, MST, Single Source Shortest Path $O(n^2)$, Matrix Multiplication $O(n^2)$.
- **Deterministic in nature**
 - A deterministic algorithm is (essentially) one that always computes the correct answer in a specific manner.
- **Solved by conventional computers in polynomial time**
- **Polynomial time upper and lower bounds exists**

NP Class Problems

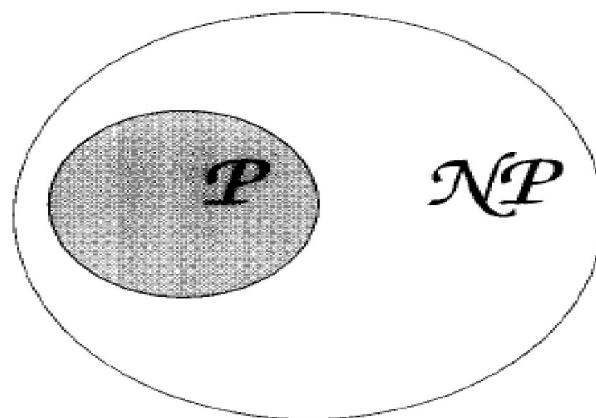
- NP (Non Deterministic Polynomial time) Class:
 - The class of problems that have no Polynomial time Solution using deterministic algorithm. Also called **Intractable Problems**.
 - The class of problems that **are solvable in polynomial time** only on a *nondeterministic* machine (or with a nondeterministic algorithm).
 - A *nondeterministic* machine is one that can “guess” the right answer or solution
 - Think of a nondeterministic computer as a **parallel machine** that can freely spawn ***an infinite number*** of processes
- Examples of NP Class problems
 - Hamiltonian Cycle $O(n2^n)$, Travelling Sales Person $O(2^n n^2)$, 0/1 Knapsack $O(2^{n-1})$, Satisfiability Problem $O(2^n)$ etc

NP Class cont..

- The class NP also consists of those problems that are **verifiable in polynomial time**.
 - If we were somehow given a "**certificate**" of a solution to a problem, then we can **verify that the certificate is correct in time polynomial**
 - **For example**, in the Hamiltonian cycle problem, given a directed graph $G(V,E)$, certificate would be a sequence $(v_1, v_2, v_3, \dots, v_n)$ of $|V|$ vertices.
 - We could easily check in polynomial time that (v_i, v_{i+1}) belongs to E for $i=1, 2, 3, \dots, |V|-1$ and that (v_n, v_1) belongs to E as well.
 - In other words it can be verified whether an HC cycle exists or not with the given sequence or not in polynomial time

P and NP

- Deterministic algorithms are just a special case of non-deterministic algorithms.
- Any problem in P is also in NP, since if a problem is in P, then we can solve it in polynomial time without even being supplied a certificate.
- We conclude that $\mathbf{P} \subseteq \mathbf{NP}$



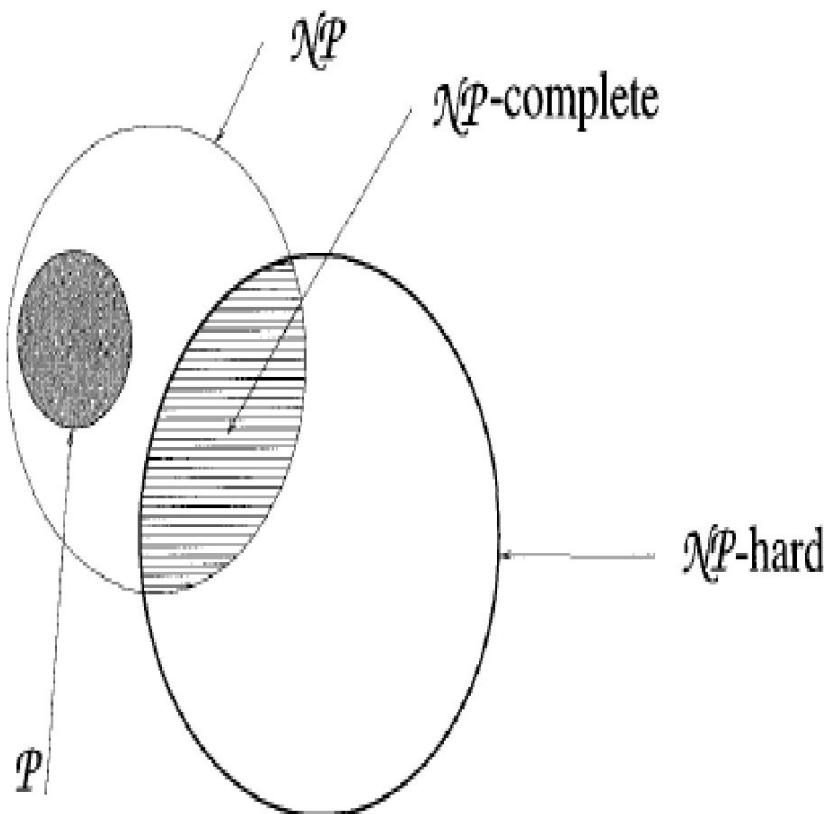
S. Cook Theorem

- If there is a problem in NP such that if we showed it to be in P , then it would imply that $P = \text{NP}$. However this is not proved. Hence concluded that $P \subseteq \text{NP}$.
- Does the problem x can present in P & NP ?
- Some conclusion is the decision problem of NP can solved in polynomial time with non-deterministic algorithm if un-limited parallel computing power is extended.
 - Ex: search problem
- But there is no conclusion that $\text{NP} = P$. Hence Cook concluded that $P \subseteq \text{NP}$

P, NP, NP-Complete, NP-Hard Type

- **P Class :** Collection of problems that can be solved by deterministic algorithms in Polynomial time
- **NP Class :** Collection of Problems that can be solved by deterministic algorithms in Non-Polynomial time (in other words) Problems that can be solved by Non-deterministic algorithms in polynomial time.
- **NP-Complete :** The decision problems that can be solved by **non-deterministic algorithm** in polynomial time.
 - It is the set of problems which are not having known algorithms that can solve in polynomial time however we can not rule out their existence.
- **NP-Hard :** The set of problems where there is no known polynomial time algorithm present to solve it.
 - If an NP-hard problem can be solved in polynomial time, then all NP-complete problems can also be solved in polynomial time.
 - All NP-complete problems are NP-hard, but all NP-hard problems are not NP-complete.

Relation between P, N, NP, NP-Complete and NP-hard



Criteria for a problem to be
NP-hard/NP-Complete

If a solution to a problem, is given, and it takes polynomial time to verify it, but cannot find the solution to a problem in Polynomial time then the problem falls in **NP class , NP Hard problem.**

Reducibility

- Let L_1 and L_2 be problems. Problem L_1 reduces to L_2 ($L_1 \propto L_2$), iff there is a way to solve L_1 by a deterministic polynomial time algorithm using a deterministic algorithm that solve L_2 In polynomial time.
- A problem L is **NP-hard** iff satisfiability reduces to L ie (satisfiability $\propto L$). A problem L is **NP-complete** iff L is **Np-hard** and $L \in NP$

Reducibility

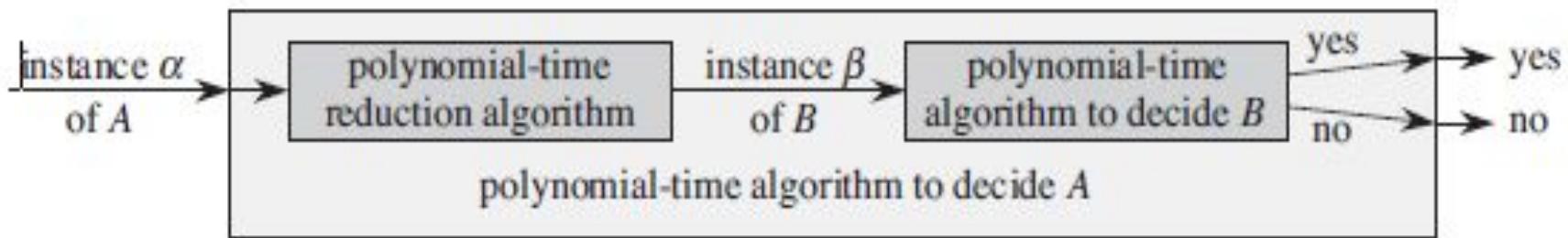


Figure 34.1 How to use a polynomial-time reduction algorithm to solve a decision problem A in polynomial time, given a polynomial-time decision algorithm for another problem B . In polynomial time, we transform an instance α of A into an instance β of B , we solve B in polynomial time, and we use the answer for β as the answer for α .

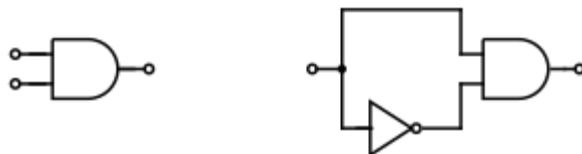
Polynomial-time *reduction algorithm*

1. Given an instance α of problem A , use a polynomial-time reduction algorithm to transform it to an instance β of problem B .
2. Run the polynomial-time decision algorithm for B on the instance β .
3. Use the answer for β as the answer for α .

In other words, by “reducing” solving problem A to solving problem B, we use the “easiness” of B to prove the “easiness” of A.

Satisfiability Problem

- It is abbreviated as SAT problem, this problem determines if there exists a set of Boolean variables which satisfy a Boolean formula ie it evaluates to **True**



- The circuit on the left is satisfiable but the circuit on the right is not.
- We try to determine the values of Boolean variables to be 1 or 0 /TRUE or False, that satisfies Boolean formula which will be evaluated to TRUE or 1.
- Boolean Formula contains
 - Variables : Truth Values **True or False**
 - Operations : It has operations **AND, OR, NOT** etc

SAT Example

Input : A Boolean Formula

Output : Find for which combination of the variables the Boolean formula become **True**

Example: For which P and Q values the formula $\sim(P \wedge Q)$ is True

P	Q	$P \wedge Q$	$\sim(P \wedge Q)$
T	T	T	F
T	F	F	T
F	T	F	T
F	F	F	T

- Therefore the $\sim(P \wedge Q)$ is satisfiable for
 - $P=F$ and $Q=F$
 - $P=F$ and $Q=T$
 - $P=F$ and $Q=F$

SAT Example

Example: Is the formula $(P \wedge \sim P)$ is satisfiable or not

Answer : No, because for any value of P , $(P \wedge \sim P)$ is not TRUE.

P	$\sim P$	$P \wedge \sim P$
T	F	F
F	T	F

Circuit Satisfiability Problem (CNF)

- Circuit Satisfiability Problem is the decision problem of determining whether this circuit has an assignment of its inputs that makes the output **TRUE**.
- To prove that the circuit-satisfiability problem is **NP complete** relies on a basic understanding of Boolean combinational circuits.
- The Boolean combinational elements that we use in the circuit-satisfiability problem computes simple Boolean functions, and they are known as **logic gates**.
 - Let $x_1, x_2, x_3 \dots$ be Boolean variables and
 - Let x_i denote negation of x_i ,
 - A literal is either x_i or \bar{x}_i ,
 - A Boolean formula is in propositional calculus constructed using literals and the operations **and**(\cap) / **or** (\cup).

CNF Satisfiability

- The formula is in CNF if and only if it is represented as $\cap_{i=1}^k c_i$, where c_i s are formulas(clauses) represented using \cup literals
- $(x_1 \cup x_2 \cup x_3) \cap (x_1 \cup x_2 \cup x_3)$

Satisfiability (SAT) problem

- Given a Boolean expression on n variables, can we assign values such that the expression is **TRUE?**

Truth Table

- $(x_1 \cup x_2 \cup x_3) \cap (x_1 \cup x_2 \cup x_3)$

	x_1	x_2	x_3
1.	0	0	0
2.	0	0	1
3.	0	1	0
4.	0	1	1
5.	1	0	0
6.	1	0	1
7.	1	1	0
8.	1	1	1

- **Answer :**

– 0 0 1 the Boolean Formula is **TRUE (satisfiable)**

– 0 1 0 the Boolean Formula is **FALSE (Not-satisfiable)**

- Time is $O(2^n)$
- Seems simple enough, but no known **deterministic polynomial time algorithm exists.**

2-CNF satisfiability

- Boolean formula has variables that can take value true or false
- The variables are connected by operators \cap , \cup and \neg
- A Boolean formula is satisfiable if there exists some assignment of values to its variables that cause it to evaluate it to true
- A Boolean formula is in k-conjunctive normal form (k-CNF) if it is the AND of clauses of ORs of exactly k variables or their negations
- 2-CNF: $(x_1 \cup x_2) \cap (x_1 \cup x_3) \cap (x_2 \cup x_3)$
 - Satisfied by $x_1 = \text{true}$, $x_2 = \text{false}$, $x_3 = \text{true}$
- We can determine in polynomial time whether a 2-CNF formula is satisfiable but satisfiability of a 3-CNF formula is NP-complete

3-CNF

- A 3-CNF is a Boolean formula that is an AND of clauses of which is an OR of exactly 3 distinct literals.
 - Ex: $(x_1 \cup x_2 \cup x_3) \cap (x_1 \cup x_2 \cup x_3)$
 - Ex2: $(x_1 + x_2 + x_3) . (x_1 + x_2 + x_3) . (x_1 + x_2 + x_3)$
- An assignment to the Boolean variables in a formula is known as truth assignment.
- A truth assignment of variables is said to be satisfying if it causes the formula to evaluate to “**True**”
- A 3-CNF is said to be satisfiable if it has a satisfying assignment.
- **What is 3-SAT Problem ?**
 - Given any Boolean formula in CNF such that each clause has exactly 3 literals, such as the one in examples. Is the formula satisfiable ?

Example

- $P1 = (x_1+x_2+x_3).(x_4+x_2+x_1).(x_3+x_2+x_1).$
 $(x_3+x_4+x_1).(x_4+x_2+x_1).(x_3+x_4+x_1).(x_4+x_2+x_3)$

					P1
F	F	F	F	F	F
F	F	F	T	F	
F	F	T	F	F	
F	F	T	T	T	
F	T	F	F	F	
F	T	F	T	F	
F	T	T	F	F	
F	T	T	T	T	

					P1
T	F	F	F	F	F
T	F	F	T	T	
T	F	T	F	T	
T	F	T	T	T	
T	T	F	F	F	
T	T	F	T	F	
T	T	T	F	F	
T	T	T	T	F	

- The Yellow Rows are assignments that makes formula TRUE. Hence P1 is Satisfiable.

Example

- $P2 = (x_1+x_2+x_3) \cdot (x_1+x_3+x_4) \cdot (x_1+x_3+x_4) \cdot (x_2+x_4+x_3) \cdot (x_1+x_2+x_3) \cdot (x_2+x_3+x_4) \cdot (x_1+x_2+x_4) \cdot (x_1+x_2+x_4)$

				P2
F	F	F	F	F
F	F	F	T	F
F	F	T	F	F
F	F	T	T	F
F	T	F	F	F
F	T	F	T	F
F	T	T	F	F
F	T	T	T	F

				P2
T	F	F	F	F
T	F	F	T	F
T	F	T	F	F
T	F	T	T	F
T	T	F	F	F
T	T	F	T	F
T	T	T	F	F
T	T	T	T	F

- The Rows have no assignments that makes formula TRUE. Hence P2 is not Satisfiable.

CNF satisfiability

Ex: $((x_1 \cup \overline{x}_2 \cup x_3) \cap (\overline{x}_1 \cup x_2 \cup \overline{x}_3))$

- A Boolean formula with n variables has 2^n Possible assignments
- If the length of clauses is polynomial in n , then checking every assignment requires $O(2^n)$ time.
- Hence, CNF Satisfiability Problem gets exponentially harder as the no. of variables increase. Hence it is in **NP Hard Problem**.
- If a Nondeterministic algorithm exists
 - Guess truth assignment
 - Check assignment to see if it satisfies CNF formula
- Given an assignment of Boolean variables, then verifying to see if it satisfies CNF Formula takes polynomial time.
 - Checking phase: $\Theta(n)$
- Easy to verify in polynomial time! When solution is given. Hence the problem is **NP Complete**.
- To show that NP-hard decision problem is NP-complete, we have just to exhibit a polynomial time nondeterministic algorithm for it.

Satisfiability

```
1 Algorithm Eval( $E$ ,  $n$ )
2 // Determine whether the propositional formula  $E$  is
3 // satisfiable. The variables are  $x_1, x_2, \dots, x_n$ .
4 {
5     for  $i := 1$  to  $n$  do // Choose a truth value assignment.
6          $x_i := \text{Choice}(\text{false}, \text{true})$ ;
7         if  $E(x_1, \dots, x_n)$  then Success();
8         else Failure();
9 }
```

Algorithm 11.6 Nondeterministic satisfiability

Examples of P-Type and NP-Type

Easy Problems (P-Type)	Hard Problems (NP-Complete)
2-SAT, Horn SAT	3-SAT
Min. Cost Spanning Tree	TSP
Fractional Knapsack	Knapsack
Shortest Path	Longest Path