19CAB02 –
Data Structures and Algorithms

UNIT-I

- Introduction - Abstract Data Types (ADT) – Arrays and its representation –Structures – Stack – Queue– Circular Queue - Applications of stack – Infix to postfix conversion – evaluation of expression – Applications of Queue - Linked Lists – Doubly Linked lists – Applications of linked list – Polynomial Addition.

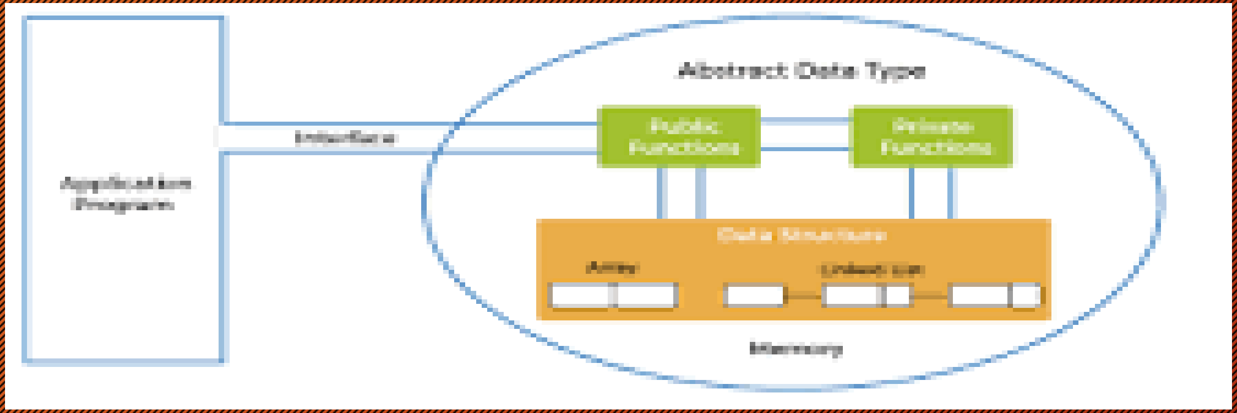
Introduction

- **Data Structure** can be defined as the group of **data** elements which provides an efficient way of storing and organizing **data** in the computer so that it can be used efficiently.
- Some examples of **Data Structures** are arrays, Linked List, Stack, Queue, etc.
- **Data Structure** is a way of organizing all **data** items that considers not only the elements stored but also. their relationship to each other.

Abstract Data Types

- **Abstract Data type (ADT)** is a type (or class) for objects whose behaviour is defined by a **set of value and a set of operations.**
- The definition of ADT only mentions **what operations are to be performed but not how these operations will be implemented.**

ADT Diagram

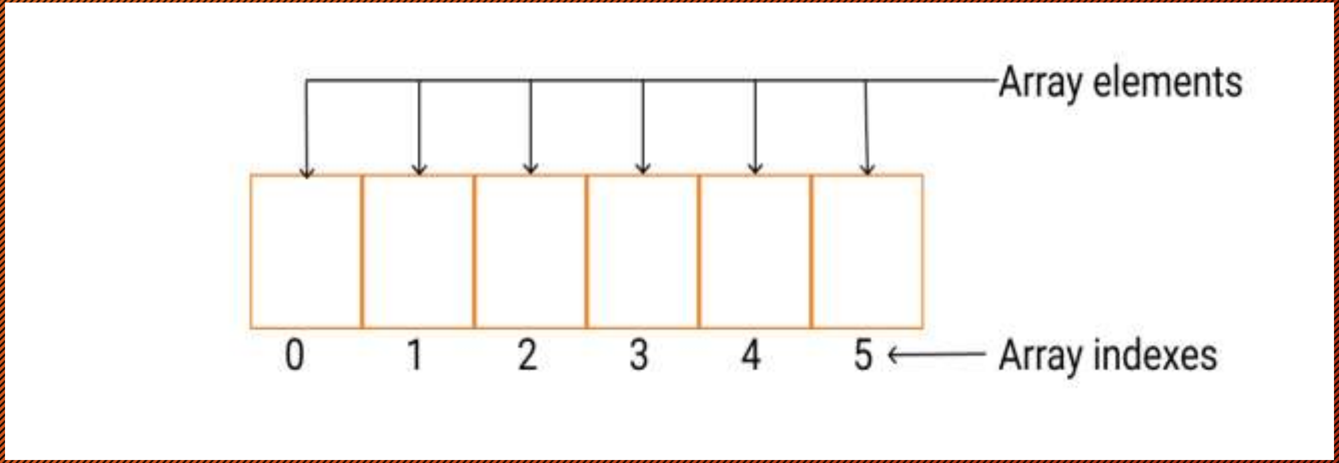


Array

- Array is a container which can hold a fix number of items and these items should be of the same type.
- Most of the data structures make use of arrays to implement their algorithms.
- Following are the important terms to understand the concept of Array.
  - **Element** – Each item stored in an array is called an element.
  - **Index** – Each location of an element in an array has a numerical index, which is used to identify the element.

Array Representation

- **Arrays and its representation** is given below **Array** Index: The location of an element in an **array** has an index, which identifies the element. **Array** index starts from 0.



Array Index: The location of an element in an array has an index, which identifies the element. Array index starts from 0.

Array element: Items stored in an array is called an element. The elements can be accessed via its index.

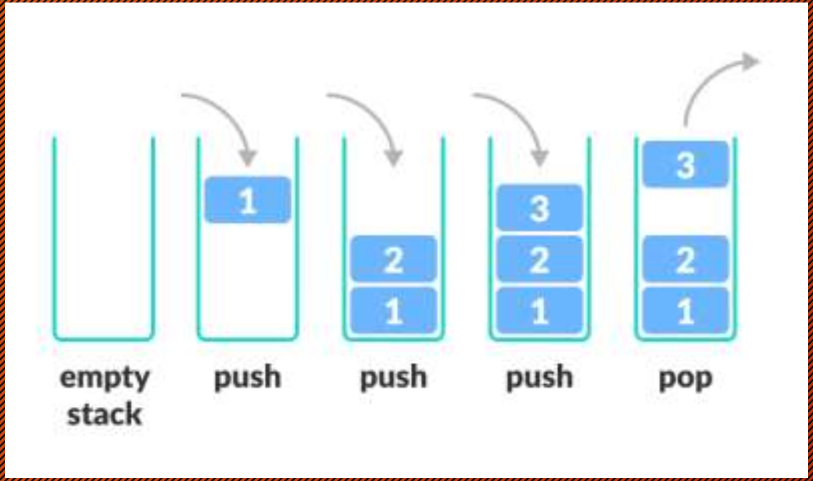
Array Length: The length of an array is defined based on the number of elements an array can store. In the above example, array length is 6 which means that it can store 6 elements.

Structures

- Data Structure can be defined as the group of data elements which provides an efficient way of storing and organizing data in the computer so that it can be used efficiently.
- The arrangement of data in a sequential manner is known as a linear data structure.
- It plays a vital role in enhancing the performance of a software or a program as the main function of the software is to store and retrieve the user's data as fast as possible.
- The data structures used for this purpose are **Arrays, Linked list, Stacks, and Queues**.

Stack

- Stack is **a linear data structure which follows a particular order in which the operations are performed.** The order may be LIFO(Last In First Out) or FILO(First In Last Out). ...
- So, it can be simply seen to follow LIFO(Last In First Out)/FILO(First In Last Out) order.



Stack Operations

- Stack operations may involve initializing the stack, using it and then de-initializing it.
- Apart from these basic stuffs, a stack is used for the following two primary operations –
- push() – Pushing (storing) an element on the stack.
- pop() – Removing (accessing) an element from the stack.
- When data is PUSHed onto stack.
- To use a stack efficiently, we need to check the status of stack as well. For the same purpose, the following functionality is added to stacks –
- peek() – get the top data element of the stack, without removing it.
- isFull() – check if stack is full.
- isEmpty() – check if stack is empty

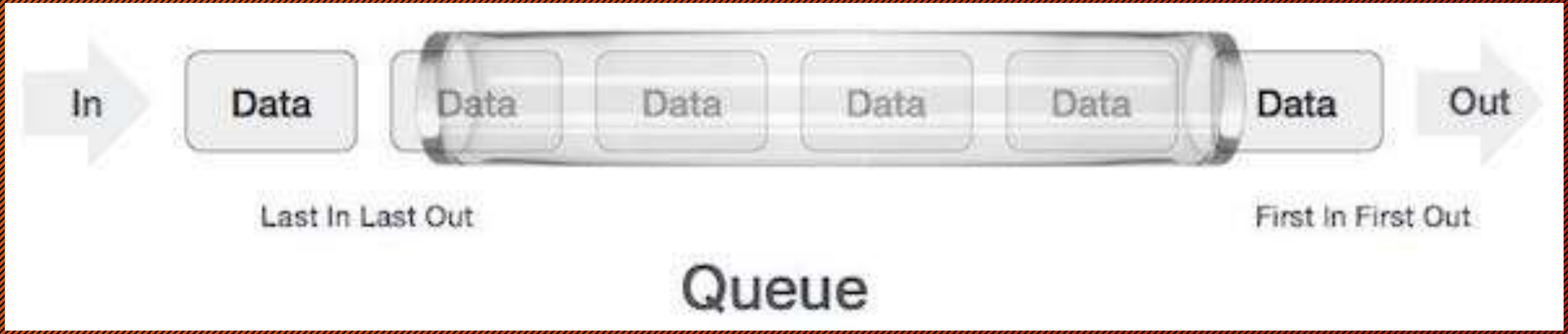
Applications of Stack

- Stacks can be used for expression evaluation.
- Stacks can be used to check parenthesis matching in an expression.
- Stacks can be used for Conversion from one form of expression to another.
- Stacks can be used for Memory Management.
- Stack data structures are used in backtracking problems

Queue

- Queue is a linear data structure where the first element is inserted from one end called REAR and deleted from the other end called as FRONT.

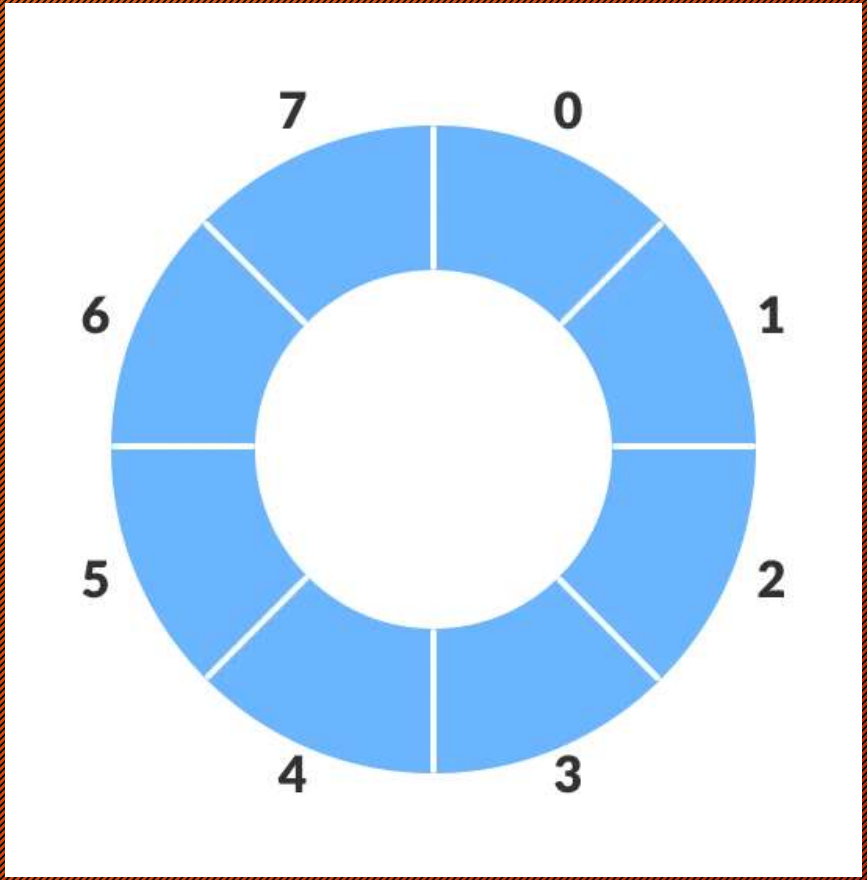
- Queue follows the FIFO (First - In - First Out) structure.



Circular Queue

- In a circular queue, all nodes are treated as circular. Last node is connected back to the first node.
- Circular queue is also called as Ring Buffer.
- It is an abstract data type.

- Circular queue contains a collection of data which allows insertion of data at the end of the queue and deletion of data at the beginning of the queue.



Infix to Postfix Conversion

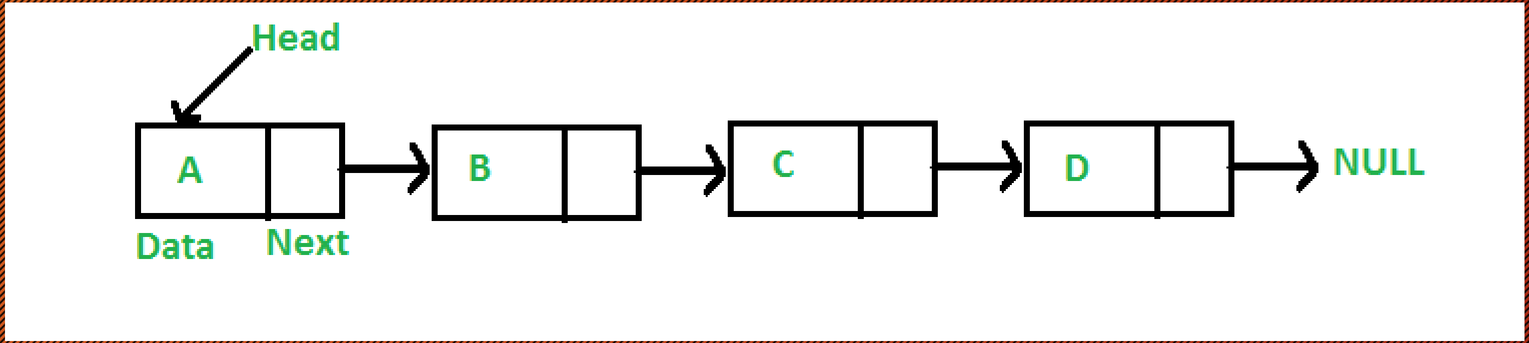
- Infix expressions are readable and solvable by humans. We can easily distinguish the order of operators, and also can use the parenthesis to solve that part first during solving mathematical expressions. The computer cannot differentiate the operators and parenthesis easily, that’s why postfix conversion is needed.
- To convert infix expression to postfix expression, we will use the stack data structure. By scanning the infix expression from left to right, when we will get any operand, simply add them to the postfix form, and for the operator and parenthesis, add them in the stack maintaining the precedence of them.

Applications of Queue

- Cashier line in any store.
- Waiting on hold for tech support.
- People on an escalator.
- Check out any book store.

Linked List

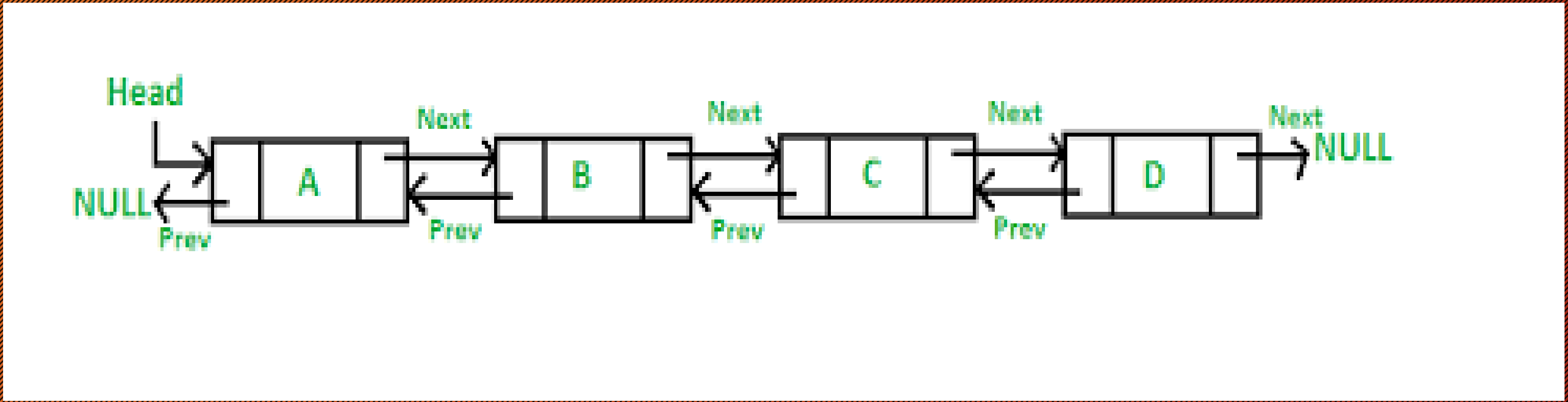
- A linked list is a sequence of data structures, which are connected together via links.
- Linked List is a sequence of links which contains items.
- Each link contains a connection to another link. Linked list is the second most-used data structure after array.



Doubly Linked List

- Doubly Linked List is a variation of Linked list in which navigation is possible in both ways, either forward and backward easily as compared to Single Linked List. Following are the important terms to understand the concept of doubly linked list.
-
- **Link** – Each link of a linked list can store a data called an element.
- **Next** – Each link of a linked list contains a link to the next link called Next.

Prev – Each link of a linked list contains a link to the previous link called Prev.
LinkedList – A Linked List contains the connection link to the first link called First and to the last link called Last.



Applications of Linked List

- Polynomial Manipulation representation.
- Addition of long positive integers.
- Representation of sparse matrices.
- Addition of long positive integers.
- Symbol table creation.
- Mailing list.
- Memory management.
- Linked allocation of files.

Polynomial Addition.

- Addition of two polynomials involves combining like terms present in the two polynomials.
-

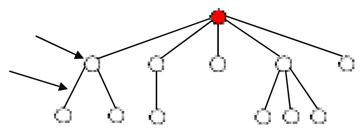
  By like terms we mean the terms having same variable and same exponent.
  For example two terms are like only if:
- The two terms have same variable
- The two terms have same power of the variable

UNIT-II

TREE STRUCTURES

Need for non-linear structures – Trees and its representation – Binary Tree – expression trees – Binary tree traversals – left child right sibling data structures for general trees – applications of trees – Huffman Algorithm - Binary search tree.

What Is a Tree?

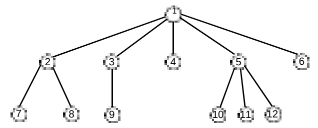


•A tree consists of:
• a set of *nodes*
• a set of *edges*, each of which connects a pair of nodes

•Each node may have one or more *data items.*
• each data item consists of one or more fields
• *key field* = the field used when searching for a data item
• multiple data items with the same key are referred to as *duplicates*

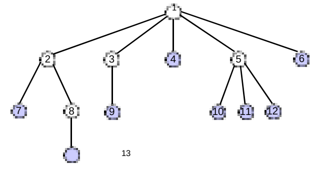
•The node at the “top” of the tree is called the *root* of the tree.

Relationships Between Nodes



- If a node N is connected to other nodes that are directly below it in the tree, N is referred to as their *parent* and they are referred to as its *children*.
  - example: node 5 is the parent of nodes 10, 11, and 12
- Each node is the child of *at most one* parent.
- Other family-related terms are also used:
- nodes with the same parent are *siblings*
- a node’s *ancestors* are its parent, its parent’s parent, etc.
  - example: node 9’s ancestors are 3 and 1
- a node’s *descendants* are its children, their children, etc.
  - example: node 1’s descendants are *all* of the other nodes

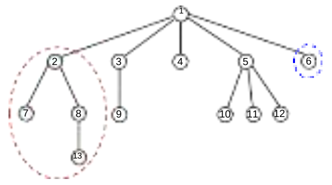
Types of Nodes



•A *leaf node* is a node without children.

•An *interior node* is a node with one or more children.

A Tree is a Recursive Data Structure

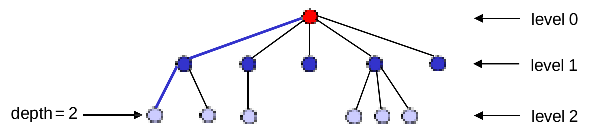


•Each node in the tree is the root of a smaller tree!

- refer to such trees as *subtrees* to distinguish them from the tree as a whole
- example: node 2 is the root of the subtree circled above
- example: node 6 is the root of a subtree with only one node

•We'll see that tree algorithms often lend themselves to recursive implementations.

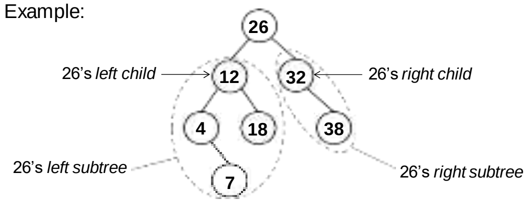
Path, Depth, Level, and Height



- There is exactly one *path* (one sequence of edges) connecting each node to the root.
- *depth* of a node = # of edges on the path from it to the root
- Nodes with the same depth form a *level* of the tree.
- The *height* of a tree is the maximum depth of its nodes.
  - example: the tree above has a height of 2

Binary Trees

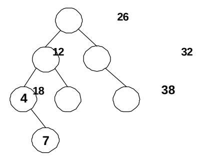
- In a *binary tree*, nodes have *at most two* children.
- Recursive definition: a binary tree is either:
  1) empty, or
  2) a node (the root of the tree) that has
     - one or more data items
     - a *left child*, which is itself the root of a binary tree
     - a *right child*, which is itself the root of a binary tree
- Example:

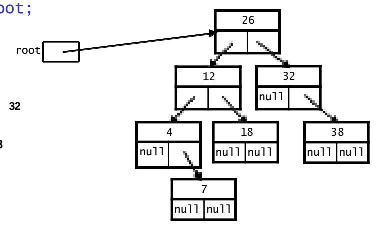


- How are the edges of the tree represented?

Representing a Binary Tree Using Linked Nodes

```
public class LinkedTree {  private class Node {
private int key;

        private LLList data;    // list of data items
        private Node left;      // reference to left child
        private Node right;     // reference to right child
        …
    }
    private Node root;
    …
}
```





• see ~cscie119/examples/trees/LinkedTree.java

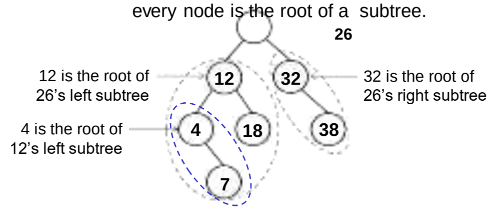
Traversing a Binary Tree

•Traversing a tree involves *visiting* all of the nodes in the tree.

- visiting a node = processing its data in some way
  - example: print the key

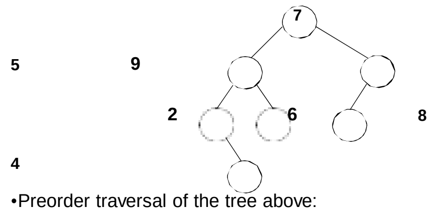
•We will look at four types of traversals. Each of them visits the nodes in a different order.

•To understand traversals, it helps to remember the recursive definition of a binary tree, in which every node is the root of a subtree.


26
12 is the root of 26's left subtree
32 is the root of 26's right subtree
4 is the root of 12's left subtree
12
32
4
18
38
7

Preorder Traversal

•preorder traversal of the tree whose root is N:
1) visit the root, N
2) recursively perform a preorder traversal of N's left subtree
3) recursively perform a preorder traversal of N's right subtree



•Preorder traversal of the tree above:

7 5 2 4 6 9 8

•Which state-space search strategy visits nodes in this order?

Implementing Preorder Traversal

```
public class LinkedTree {
…
private Node root;
public void preorderPrint() {  if (root !=
null)
preorderPrintTree(root);
}
private static void preorderPrintTree(Node
root) {  System.out.print(root.key + "
  ");
if (root.left != null)
preorderPrintTree(root.left);  if
(root.right != null)
}       preorderPrintTree(root.right);
}
```

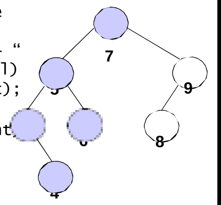
Not always the same as the root of the entire tree.

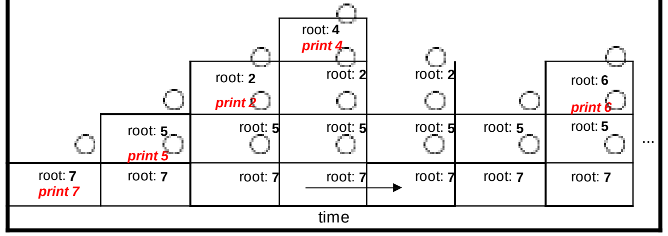
•`preorderPrintTree()` is a static, recursive method that takes as a parameter the root of the tree/subtree that you want to print.

•`preorderPrint()` is a non-static method that makes the initial call. It passes in the root of the entire tree as the parameter.

Tracing Preorder Traversal

```
void preorderPrintTree(Node
root) {
System.out.print(root.key + "
  ");  if (root.left != null)
preorderPrintTree(root.left);
if (root.right != null)
preorderPrintTree(root.right
}
```

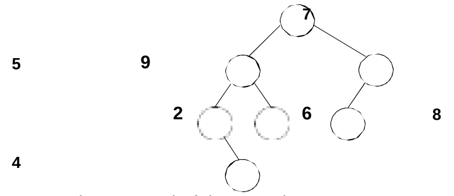




Postorder Traversal

•postorder traversal of the tree whose root is N:

1) recursively perform a postorder traversal of N's left subtree
2) recursively perform a postorder traversal of N's right subtree
3) visit the root, N



•Postorder traversal of the tree above:

4 2 6 5 8 9 7

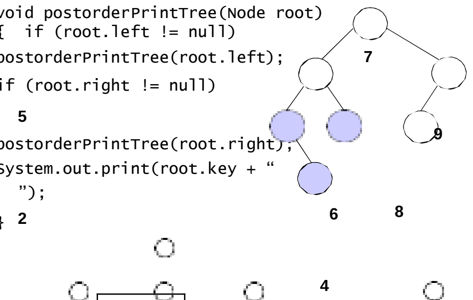
Implementing Postorder Traversal

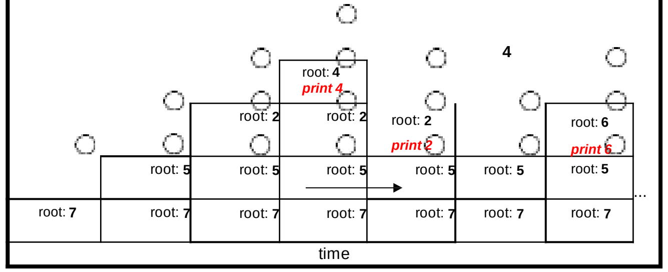
```
public class LinkedTree {
…
private Node root;

public void postorderPrint() {  if (root != null)
postorderPrintTree(root);
}

private static void postorderPrintTree(Node root) {  if (root.left
!= null)
postorderPrintTree(root.left);
if (root.right != null)  postorderPrintTree(root.right);
System.out.print(root.key + "           ");
}
}
```

•Note that the root is printed *after* the two recursive calls.

Tracing Postorder Traversal

```
void postorderPrintTree(Node root)
{  if (root.left != null)
postorderPrintTree(root.left);
if (root.right != null)
postorderPrintTree(root.right);
System.out.print(root.key + "
  ");
}
```

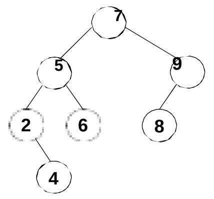




Inorder Traversal

•inorder traversal of the tree whose root is N:

1) recursively perform an inorder traversal of N's left subtree
2) visit the root, N
3) recursively perform an inorder traversal of N's right subtree



- Inorder traversal of the tree above:

  2 4 5 6 7 8 9

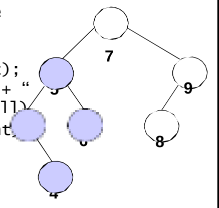
Implementing Inorder Traversal

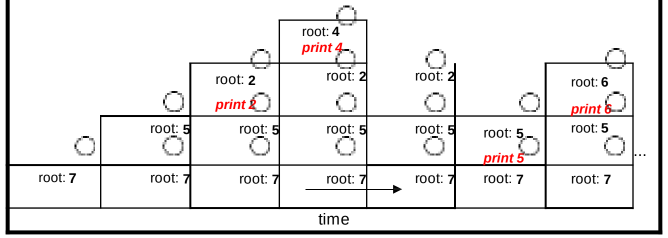
```
public class LinkedTree {
…
private Node root;

public void inorderPrint() {  if (root != null)
inorderPrintTree(root);
}

private static void inorderPrintTree(Node root) {
if (root.left != null)
inorderPrintTree(root.left);
System.out.print(root.key + “          ");  if
(root.right != null)
inorderPrintTree(root.right);
}
}
```

•Note that the root is printed *between* the two recursive calls.

Tracing Inorder Traversal

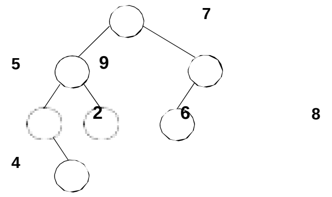
```
void inorderPrintTree(Node
root) {  if (root.left !=
null)
inorderPrintTree(root.left);
System.out.print(root.key + "
 ");  if (root.right != null)
inorderPrintTree(root.right

}
```





Level-Order Traversal

•Visit the nodes one level at a time, from top to bottom and left to right.



•Level-order traversal of the tree above: **7 5 9 2 6 8 4**

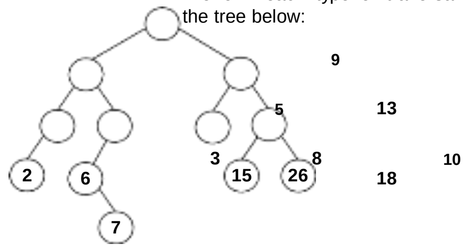
•Which state-space search strategy visits nodes in this order?

•How could we implement this type of traversal?

Tree-Traversal Summary

preorder: root, left subtree, right subtree postorder: left subtree, right subtree, root inorder: left subtree, root, right subtree level-order: top to bottom, left to right

•Perform each type of traversal on the tree below:

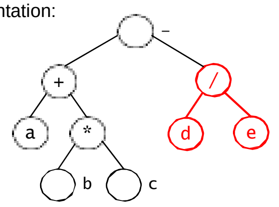


Using a Binary Tree for an Algebraic Expression

•We'll restrict ourselves to fully parenthesized expressions and to the following binary operators: +, –, *, /

•Example expression: `((a + (b * c)) – (d / e))`

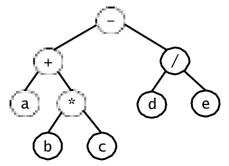
•Tree representation:



•Leaf nodes are variables or constants; interior nodes are operators.

•Because the operators are binary, either a node has two children or it has none.

Traversing an Algebraic-Expression Tree



- Inorder gives conventional algebraic notation.
  - print '(' before the recursive call on the left subtree
  - print ')' after the recursive call on the right subtree
  - for tree at right: `((a + (b * c)) – (d / e))`
  - Preorder gives functional notation.
  - print '('s and ')'s as for inorder, and commas after the recursive call on the left subtree
  - for tree above: `subtr(add(a, mult(b, c)), divide(d, e))`
  - Postorder gives the order in which the computation must be carried out on a stack/RPN calculator.
  - for tree above: `push a, push b, push c, multiply, add,…`
  - see `~cscie119/examples/trees/ExprTree.java`

Fixed-Length Character Encodings

•A character encoding maps each character to a number.

•Computers usually use fixed-length character encodings.

- ASCII (American Standard Code for Information Interchange) uses 8 bits per character.

char	dec	binary
a	97	01100001
b	98	01100010
c	99	01100011
…	…	…

example: "bat" is stored in a text file as the following sequence of bits:
01100010 01100001 01110100

- Unicode uses 16 bits per character to accommodate foreign-language characters. (ASCII codes are a subset.)

•Fixed-length encodings are simple, because

- all character encodings have the same length
- a given character always has the same encoding

Variable-Length Character Encodings

- Problem: fixed-length encodings waste space.
- Solution: use a variable-length encoding.
  - use encodings of different lengths for different characters
  - assign shorter encodings to frequently occurring characters
- Example:

e	01
o	100
s	111
t	00

"test" would be encoded as
`00 01 111 00` → `000111100`

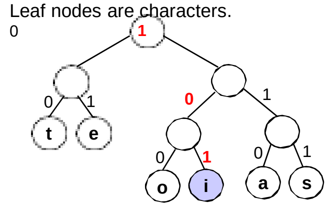
- Challenge: when decoding/decompressing an encoded document, how do we determine the boundaries between characters?
  - example: for the above encoding, how do we know whether the next character is 2 bits or 3 bits?
- One requirement: no character's encoding can be the prefix of another character's encoding (e.g., couldn't have 00 and 001).

Huffman Encoding

•Huffman encoding is a type of variable-length encoding that is based on the actual character frequencies in a given document.

•Huffman encoding uses a binary tree:
- to determine the encoding of each character
- to decode an encoded file – i.e., to decompress a compressed file, putting it back into ASCII

•Example of a Huffman tree (for a text with only six chars):
Leaf nodes are characters.



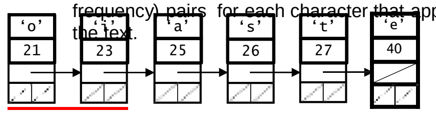
Left branches are labeled with a 0, and right branches are labeled with a 1.

If you follow a path from root to leaf, you get the encoding of the character in the leaf
example: 101 = 'i'

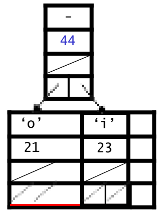
Building a Huffman Tree

1)Begin by reading through the text to determine the frequencies.

2)Create a list of nodes that contain (character, frequency) pairs for each character that appears in the text.

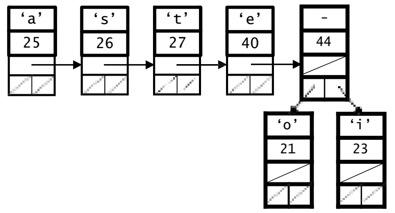


3) Remove and "merge" the nodes with the two lowest frequencies, forming a new node that is their parent.
   - left child = lowest frequency node
   - right child = the other node
   - frequency of parent = sum of the frequencies of its children
     - in this case, 21 + 23 = 44



Building a Huffman Tree (cont.)

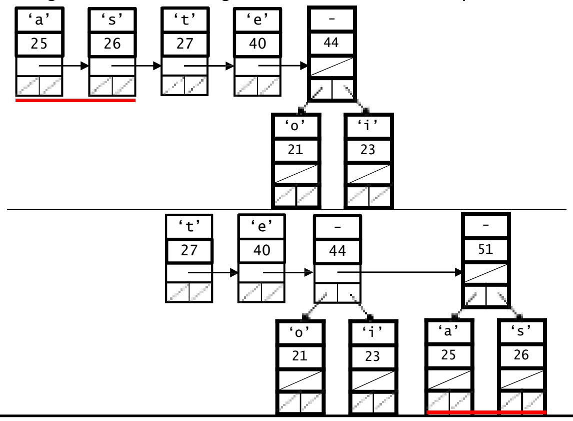
4) Add the parent to the list of nodes:



5) Repeat steps 3 and 4 until there is only a single node in the list, which will be the root of the Huffman tree.

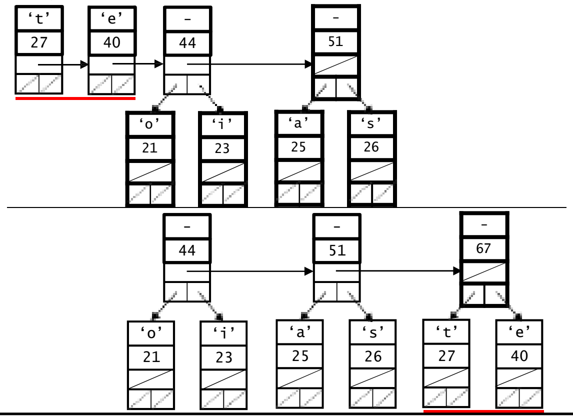
Completing the Huffman Tree Example I

- Merge the two remaining nodes with the lowest frequencies:



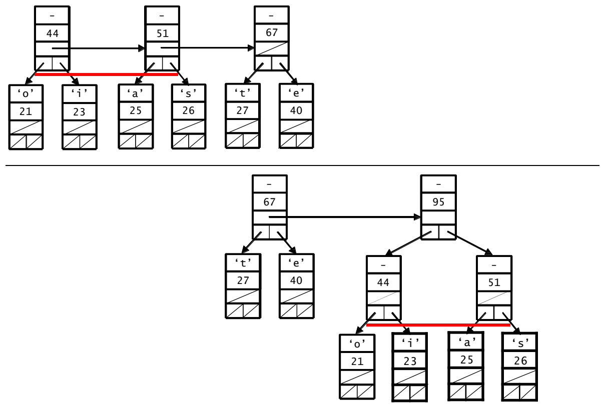
Completing the Huffman Tree Example II

- Merge the next two nodes:



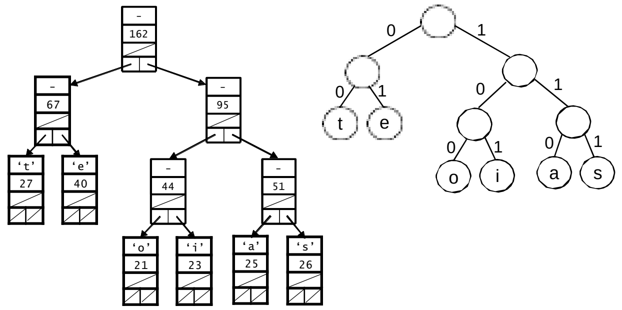
Completing the Huffman Tree Example III

- Merge again:



Completing the Huffman Tree Example IV

- The next merge creates the final tree:



- Characters that appear more frequently end up higher in the tree, and thus their encodings are shorter.

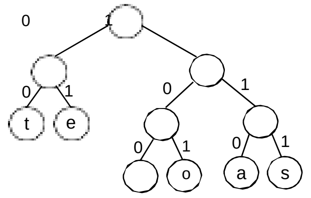
Using Huffman Encoding to Compress a File

1)Read through the input file and build its Huffman tree.

2)Write a file header for the output file.

– include an array containing the frequencies so that the tree can be rebuilt when the file is decompressed.

3)Traverse the Huffman tree to create a table containing the encoding of each character:



a	**?**
e	**?**
i	101
o	100
s	111
t	00

4) Read through the input file a second time, and write the Huffman code for each character to the output file.

Using Huffman Decoding to Decompress a File

1)Read the frequency table from the header and rebuild the tree.

2)Read one bit at a time and traverse the tree, starting from the root:

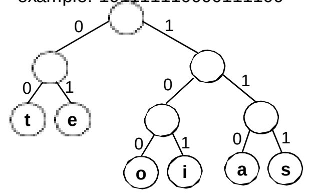
when you read a bit of 1, go to the right child

when you read a bit of 0, go to the left child

when you reach a leaf node, record the character, return to the root, and continue reading bits

The tree allows us to easily overcome the challenge of determining the character boundaries!

example: 101111110000111100

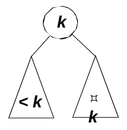


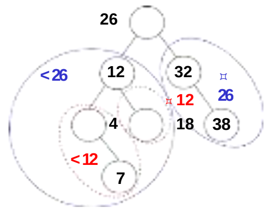
101 = right,left,right = i
111 = right,right,right= s
110 = right,right,left = a 00
= left,left = t
01 = left,right = e
111 = right,right,right= s
00 = left,left = t

Binary *Search* Trees

•Search-tree property: for each node *k:*

- all nodes in *k*'s left subtree are < *k*
- all nodes in *k*'s right subtree are >= *k*
- Our earlier binary-tree example is a search tree:

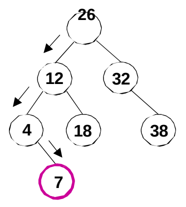




Searching for an Item in a Binary Search Tree

•Algorithm for searching for an item with a key *k*: if *k* == the root node's key, you're done
else if *k* < the root node's key, search the left subtree else search the right subtree

•Example: search for 7



Implementing Binary-Tree Search

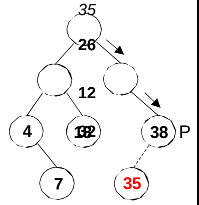
```
public class LinkedTree { private Node root; // Nodes have keys that are ints
…
public LLList search(int key) {  Node n =
searchTree(root, key);
return (n == null ? null : n.data);
}

private static Node searchTree(Node root, int key) {
// write together









}
}
```

•If we find a node that has the specified key, we return its data field, which holds a list of the data items for that key.

Inserting an Item in a Binary Search Tree

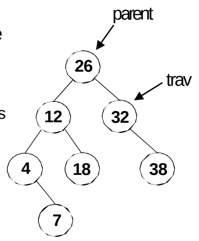
- We want to insert an item whose key is *k*.
- We traverse the tree as if we were searching for *k*.
- If we find a node with key *k*, we add the data item to the list of items for that node.
- If we don't find it, the last node we encounter will be the parent P of the new node.
  - if *k* < P's key, make the new node P's left child
  - else make the node P's right child
- *Special case:* if the tree is empty,
make the new node the root of the tree.
- The resulting tree is still a search tree.

example: insert 35



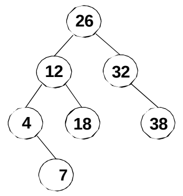
Implementing Binary-Tree Insertion

- We'll implement part of the `insert()` method together.
- We'll use iteration rather than recursion.
- Our method will use two references/pointers:
  - `trav`: performs the traversal down to the point of insertion
  - `parent`: stays one behind `trav`
    - like the `trail` reference that we sometimes use when traversing a linked list



Implementing Binary-Tree Insertion

```
      public void insert(int key, Object data)
      {  Node parent = null;
  Node trav = root;  while (trav !=
  null) {
  if (trav.key == key) {
  trav.data.addItem(data, 0);  return;
  }
```

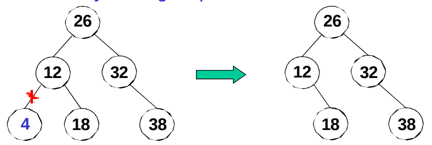


```
}
Node newNode = new Node(key, data);
if (parent == null)        // the tree was empty
root = newNode;
else if (key < parent.key)
parent.left = newNode;  else
parent.right = newNode;
}
```

Deleting Items from a Binary Search Tree

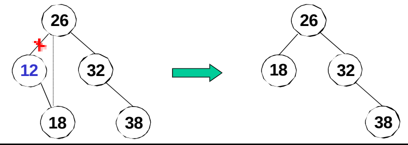
- Three cases for deleting a node *x*
- **Case 1:** *x* has no children.
  Remove *x* from the tree by setting its parent's reference to null.

ex: delete 4



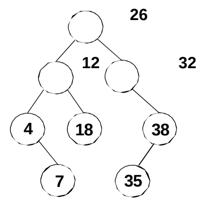
- **Case 2:** *x* has one child.
  Take the parent's reference to *x* and make it refer to *x*'s child.

ex: delete 12



Deleting Items from a Binary Search Tree (cont.)

- **Case 3:** *x* has two children
- we can't just delete *x*. why?
- instead, we replace *x* with a node from elsewhere in the tree
- to maintain the search-tree property, we must choose the replacement carefully
  - example: what nodes could replace 26 below?



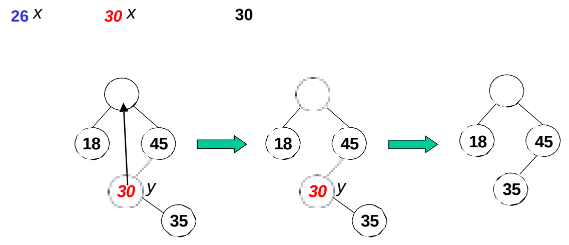
Deleting Items from a Binary Search Tree (cont.)

•**Case 3:** *x* has two children (continued):

• replace *x* with the smallest node in *x*'s right subtree— call it *y*

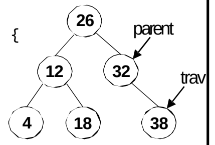
– *y* will either be a leaf node or will have one right child. why?

•After copying *y*'s item into *x*, we delete *y* using case 1 or 2. ex: delete 26



Implementing Binary-Tree Deletion

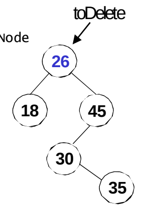
```
      public LLList delete(int key) {
      // Find the node and its parent.  Node
      parent = null;
Node trav = root;
while (trav != null && trav.key != key) {
parent = trav;
if (key < trav.key)
trav = trav.left;  else

        trav = trav.right;
}

// Delete the node (if any) and return the removed items.
if (trav == null)      // no such key
return null;
else {
LLList removedData = trav.data;  deleteNode(trav, parent);
return removedData;
}
}
```



•This method uses a helper method to delete the node.

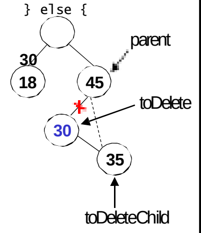
Implementing Case 3

```
                     private void deleteNode(Node toDelete,
                     Node parent) {
                     if (toDelete.left != null &&
                     toDelete.right != null) {
                     // Find a replacement – and
                     // the replacement's parent.  Node
// Get the smallest item replaceParent = toDelete;
// in the right subtree.
Node replace = toDelete.right;
// What should go here?



// Replace toDelete's key and data
// with those of the replacement item.
toDelete.key = replace.key;
toDelete.data = replace.data;

// Recursively delete the replacement
// item's old node. It has at most one
// child, so we don't have to
// worry about infinite recursion.
deleteNode(replace, replaceParent);
                                  } else {
                                       ...
}
```



Implementing Cases 1 and 2

```
  private void deleteNode(Node toDelete, Node parent)
  {
  if (toDelete.left != null && toDelete.right != null)
  {
                                             ...
                                        } else {
  Node toDeleteChild;
  if (toDelete.left != null)
      toDeleteChild = toDelete.left;
    toDeleteChild = toDelete.right;
  else
// Note: in case 1, toDeleteChild
// will have a value of null.

if (toDelete == root)  root =
    toDeleteChild;
else if (toDelete.key < parent.key)
    parent.left = toDeleteChild;
else
parent.right = toDeleteChild;
    }
}
```



Efficiency of a Binary Search Tree

•The three key operations (search, insert, and delete) all have the same time complexity.

- insert and delete both involve a search followed by a constant number of additional operations

•Time complexity of searching a binary search tree:

- best case: *O*(1)
- worst case: *O*(*h*), where *h* is the height of the tree
- average case: *O*(*h*)

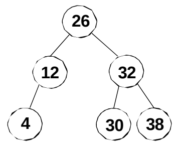
•What is the height of a tree containing n items?

- it depends! why?

Balanced Trees

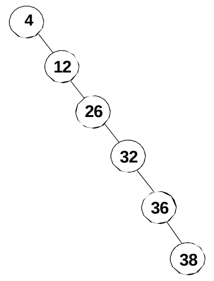
•A tree is *balanced* if, for each node, the node's subtrees have the same height or have heights that differ by 1.

- For a balanced tree with n nodes:
  - height = *O*($\log_2 n$).
  - gives a worst-case time complexity that is logarithmic (*O*($\log_2 n$))
    - the best worst-case time complexity for a binary tree



What If the Tree Isn't Balanced?

- Extreme case: the tree is equivalent to a linked list
  - height = n − 1
  - worst-case
    time complexity = *O*(n)
- We'll look next at search-tree variants that take special measures to ensure balance.



UNIT-III
Graphs

Definitions – Representation of graph - Graph Traversals - Depth-first traversal – Breadth-first traversal - Applications of graphs - Topological sort – Shortest-path algorithms – Minimum spanning tree – Prim's and Kruskal's algorithms – Biconnectivity – Euler circuits.

Where We Are

We have learned about the essential ADTs and data structures:

- Regular and Circular Arrays (dynamic sizing)
- Linked Lists
- Stacks, Queues, Priority Queues
- Heaps
- Unbalanced and Balanced Search Trees

We have also learned important algorithms

- Tree traversals
- Floyd's Method
- Sorting algorithms

Where We Are Going

Less generalized data structures and ADTs

More on algorithms and related problems that require constructing data structures to make the solutions efficient

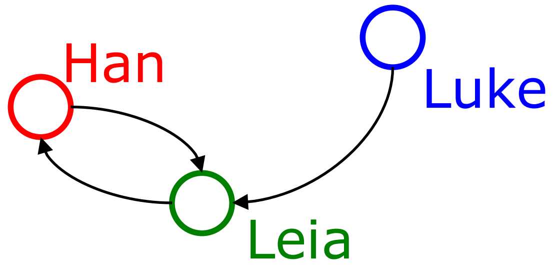
Topics will include:

- Graphs
- Parallelism

Graphs

A graph is a formalism for representing relationships among items

- Very general definition
- Very general concept



A graph is a pair: G = (V, E)

- A set of vertices, also known as nodes: $V = \{v_1, v_2, \ldots, v_n\}$
- A set of edges $E = \{e_1, e_2, \ldots, e_m\}$
  - Each edge e_i is a pair of vertices (v_j, v_k)
  - An edge "connects" the vertices

$$V = \{\text{Han}, \text{Leia}, \text{Luke}\}$$
$$E = \{(\text{Luke}, \text{Leia}), (\text{Han}, \text{Leia}), (\text{Leia}, \text{Han})\}$$

Graphs can be *directed* or *undirected*

A Graph ADT?

We can think of graphs as an ADT

- Operations would inlude **isEdge(v_j,v_k)**
- But it is unclear what the "standard operations" would be for such an ADT

Instead we tend to develop algorithms over graphs and then use data structures that are efficient for those algorithms

Many important problems can be solved by:

1. Formulating them in terms of graphs
2. Applying a standard graph algorithm

Some Graphs

For each example, what are the vertices and what are the edges?

- Web pages with links
- Facebook friends
- "Input data" for the Kevin Bacon game
- Methods in a program that call each other
- Road maps
- Airline routes
- Family trees
- Course pre-requisites

Core algorithms that work across such domains is why we are CSE

Scratching the Surface

Graphs are a powerful representation and have been studied deeply

Graph theory is a major branch of research in combinatorics and discrete mathematics

Every branch of computer science involves graph theory to some extent

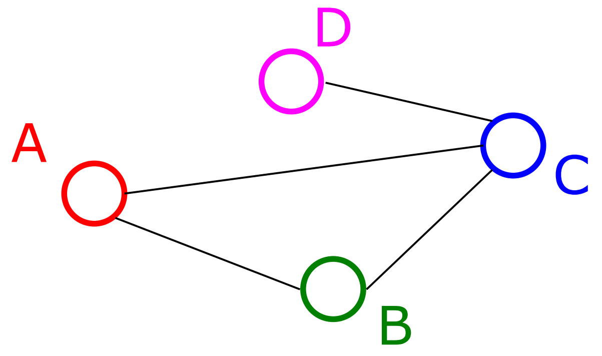
To make formulating graphs easy and standard, we have a lot of *standard terminology* for graphs

GRAPH TERMINOLOGY

Undirected Graphs

In undirected graphs, edges have no specific direction

- Edges are always "two-way"



Thus, $(u, v) \in E$ implies $(v, u) \in E$.

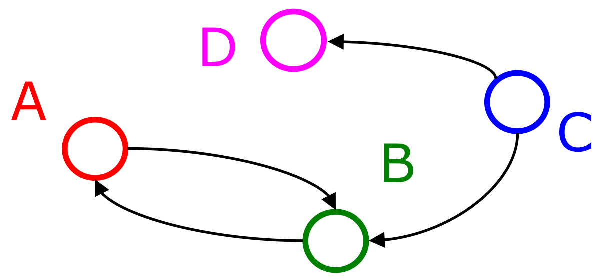
- Only one of these edges needs to be in the set
- The other is implicit, so normalize how you check for it

Degree of a vertex: number of edges containing that vertex

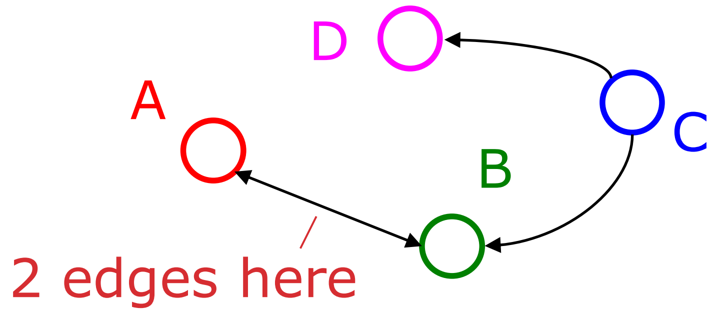
- Put another way: the number of adjacent vertices

Directed Graphs

In directed graphs (or digraphs), edges have direction



or



Thus, $(u, v) \in E$ does not imply $(v, u) \in E$.

Let $(u, v) \in E$ mean $u \rightarrow v$

- Call u the source and v the destination
- In-Degree of a vertex: number of in-bound edges (edges where the vertex is the destination)
- Out-Degree of a vertex: number of out-bound edges (edges where the vertex is the source)

Self-Edges, Connectedness

A self-edge a.k.a. a loop edge is of the form (u, u)

- The use/algorithm usually dictates if a graph has:
  - No self edges
  - Some self edges
  - All self edges

A node can have a(n) degree / in-degree / out-degree of zero

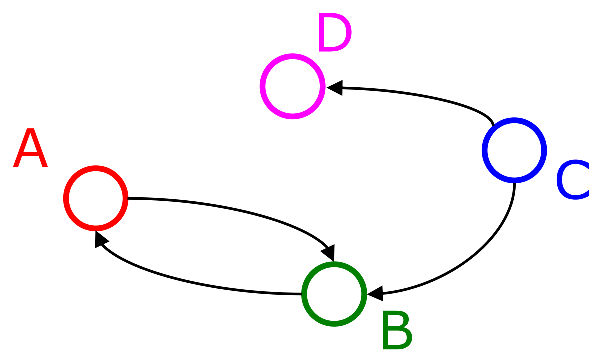
A graph does not have to be connected

- Even if every node has non-zero degree
- More discussion of this to come

More Notation

For a graph $G = (V, E)$:

- $|V|$ is the number of vertices
- $|E|$ is the number of edges
  - Minimum?
  - Maximum for undirected?
  - Maximum for directed?



$V = \{A, B, C, D\}$
$E = \{(C, B), (A, B), (B, A), (C, D)\}$

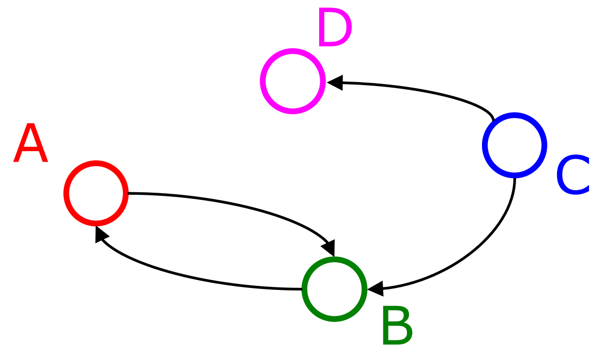
If $(u, v) \in E$, then v is a neighbor of u (i.e., v is adjacent to u)

- Order matters for directed edges:
u is not adjacent to v unless $(v, u) \in E$

More Notation

For a graph $G = (V, E)$:

- $|V|$ is the number of vertices
- $|E|$ is the number of edges
  - Minimum? 0
  - Maximum for undirected? $|V||V+1|/2 \in O(|V|^2)$
  - Maximum for directed? $|V|^2 \in O(|V|^2)$



If $(u, v) \in E$, then v is a neighbor of u (i.e., v is adjacent to u)

- Order matters for directed edges:

u is not adjacent to v unless $(v, u) \in E$

Examples Again

Which would use directed edges?

Which would have self-edges?

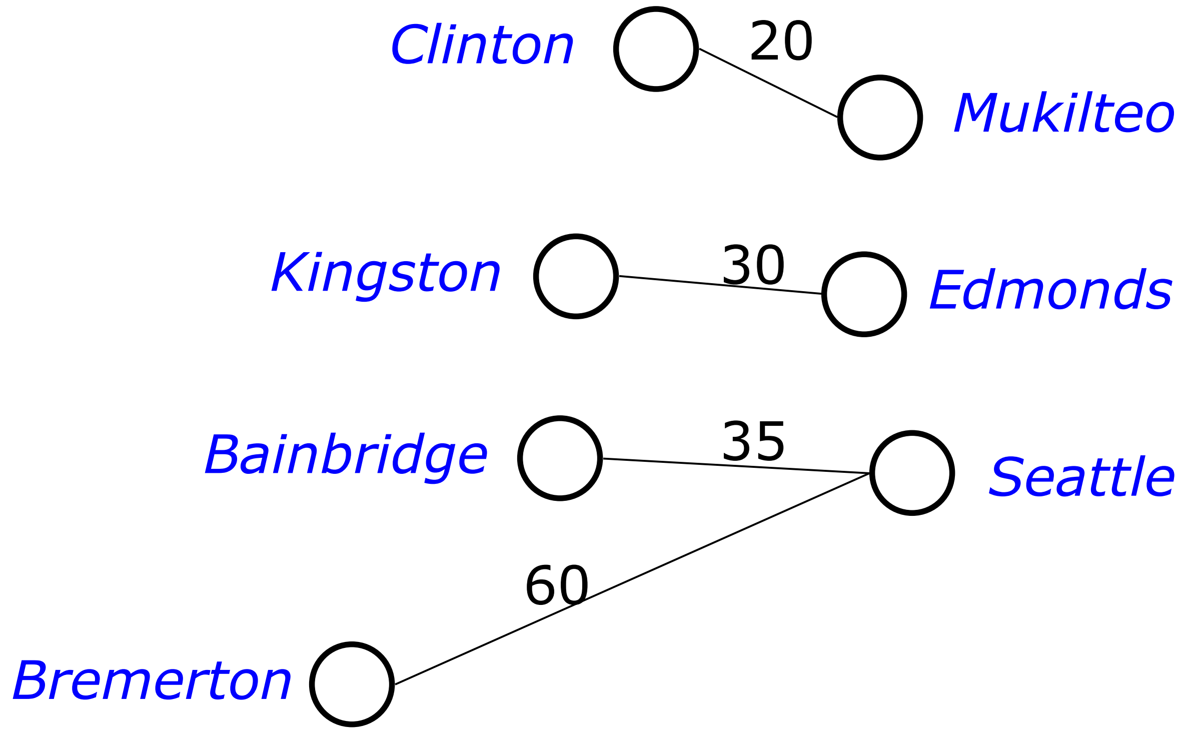
Which could have 0-degree nodes?

- Web pages with links
- Facebook friends
- "Input data" for the Kevin Bacon game
- Methods in a program that call each other
- Road maps
- Airline routes
- Family trees
- Course pre-requisites

Weighted Graphs

In a weighted graph, each edge has a weight or cost

- Typically numeric (ints, decimals, doubles, etc.)
- Orthogonal to whether graph is directed
- Some graphs allow negative weights; many do not



Examples Again

What, if anything, might weights represent for each of these?

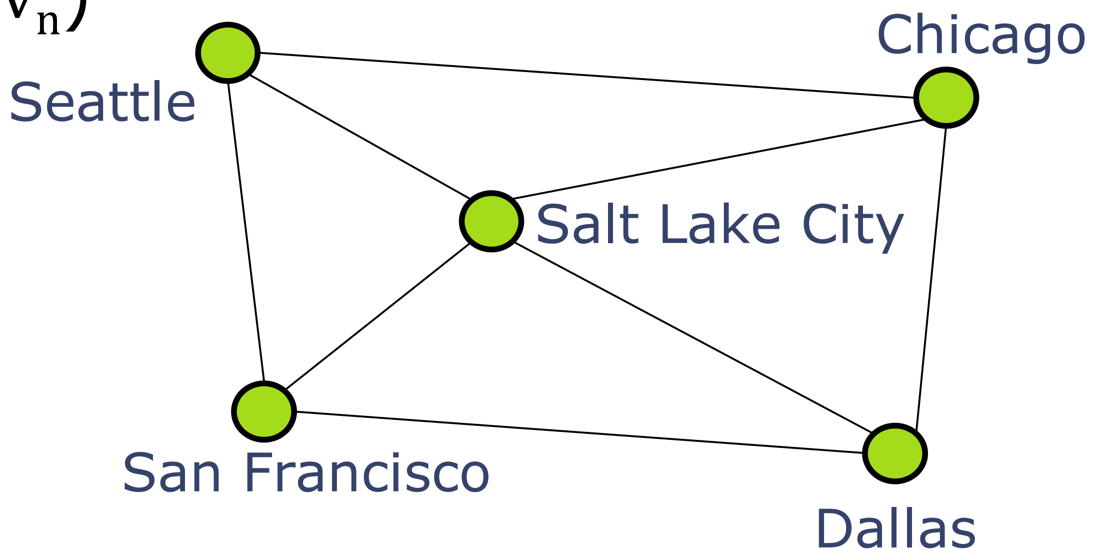
Do negative weights make sense?

- Web pages with links
- Facebook friends
- "Input data" for the Kevin Bacon game
- Methods in a program that call each other
- Road maps
- Airline routes
- Family trees
- Course pre-requisites

Paths and Cycles

We say "a path exists from v_0 to v_n" if there is a list of vertices $[v_0, v_1, \ldots, v_n]$ such that $(v_i, v_{i+1}) \in E$ for all $0 \leq i < n$.

A cycle is a path that begins and ends at the same node ($v_0 == v_n$)



Example path (that also happens to be a cycle):

[Seattle, Salt Lake City, Chicago, Dallas, San Francisco, Seattle]

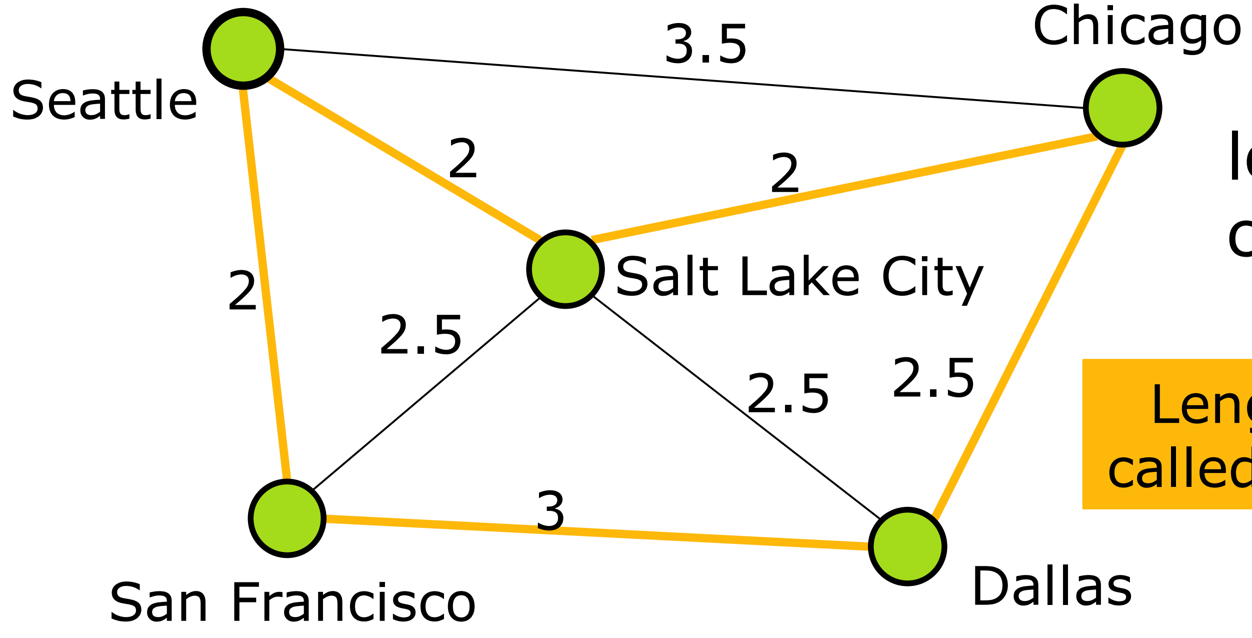
Path Length and Cost

Path length: Number of edges in a path

Path cost: Sum of the weights of each edge

Example where

P= [Seattle, Salt Lake City, Chicago, Dallas,
San Francisco, Seattle]



length(**P**) = 5
cost(**P**) = 11.5

Length is sometimes called "unweighted cost"

Simple Paths and Cycles

A simple path repeats no vertices (except the first might be the last):

[Seattle, Salt Lake City, San Francisco, Dallas]

[Seattle, Salt Lake City, San Francisco, Dallas, Seattle]

A cycle is a path that ends where it begins:

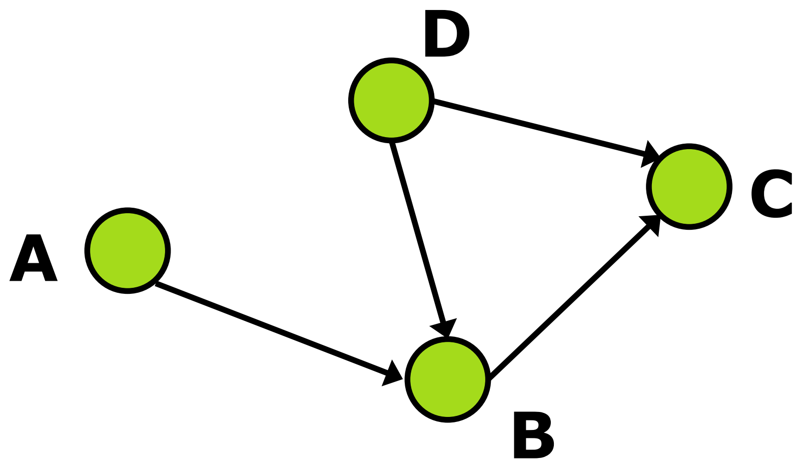
[Seattle, Salt Lake City, Seattle, Dallas, Seattle]

A simple cycle is a cycle and a simple path:

[Seattle, Salt Lake City, San Francisco, Dallas, Seattle]

Paths and Cycles in Directed Graphs

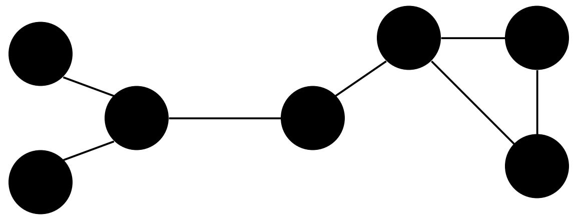
Example:

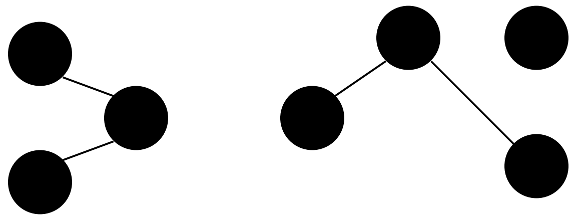


- Is there a path from A to D? No
- Does the graph contain any cycles? No

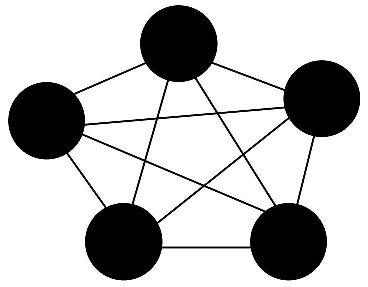
Undirected Graph Connectivity

An undirected graph is connected if for all pairs of vertices $u \neq v$, there exists a *path* from u to v


Connected graph

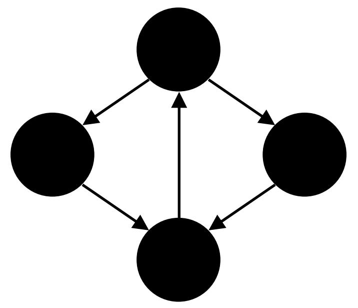

Disconnected graph

An undirected graph is complete, or fully connected, if for all pairs of vertices $u \neq v$ there exists an *edge* from u to v

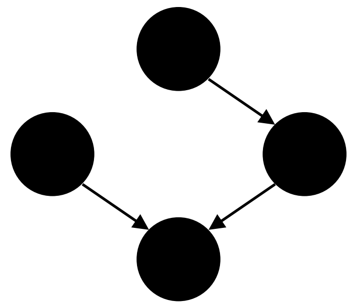


Directed Graph Connectivity

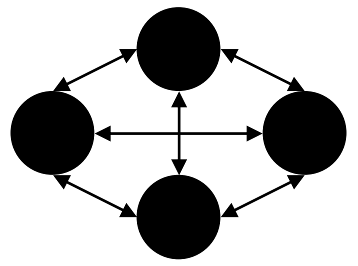
A directed graph is strongly connected if there is a path from every vertex to every other vertex



A directed graph is weakly connected if there is a path from every vertex to every other vertex *ignoring direction of edges*



A direct graph is complete or fully connected, if for all pairs of vertices $u \neq v$, there exists an *edge* from u to v



Examples Again

For undirected graphs: connected?
For directed graphs: strongly connected?
weakly connected?

- Web pages with links
- Facebook friends
- "Input data" for the Kevin Bacon game
- Methods in a program that call each other
- Road maps
- Airline routes
- Family trees
- Course pre-requisites

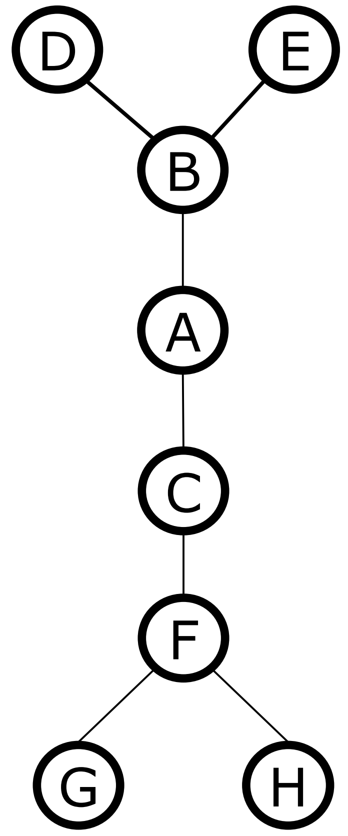
Trees as Graphs

When talking about graphs, we say a tree is a graph that is:

- undirected
- acyclic
- connected

All trees are graphs, but NOT all graphs are trees

How does this relate to the trees we know and "love"?



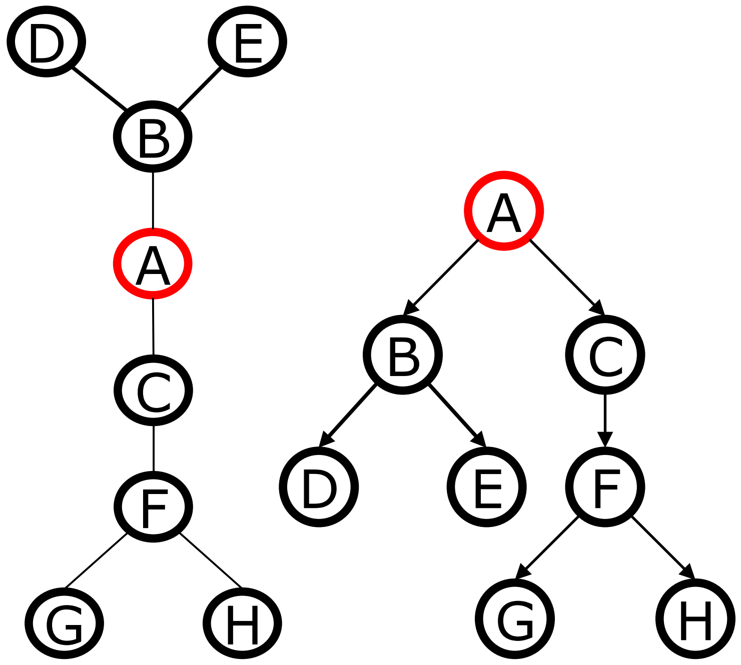
Rooted Trees

We are more accustomed to rooted trees where:

- We identify a unique root
- We think of edges as directed: parent to children

Picking a root gives a unique rooted tree

- The tree is simply drawn differently and with undirected edges



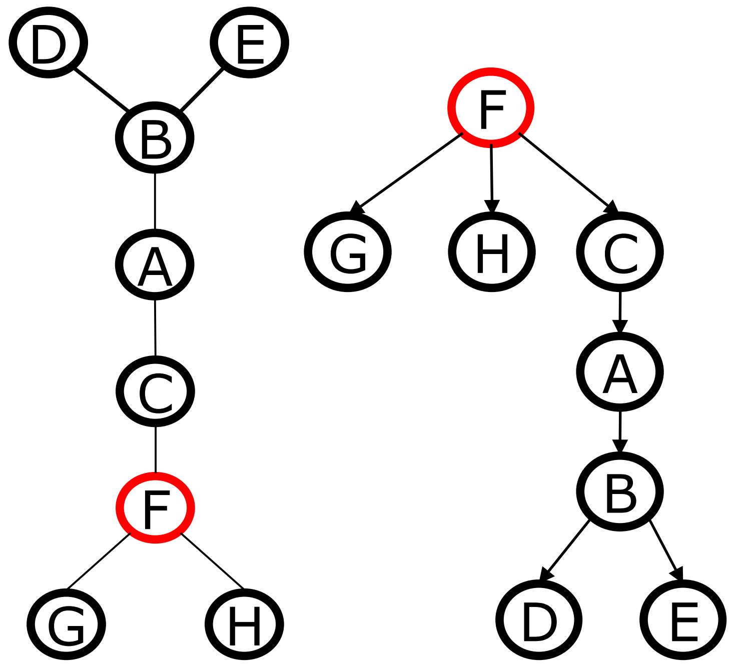
Rooted Trees

We are more accustomed to rooted trees where:

- We identify a unique root
- We think of edges as directed: parent to children

Picking a root gives a unique rooted tree

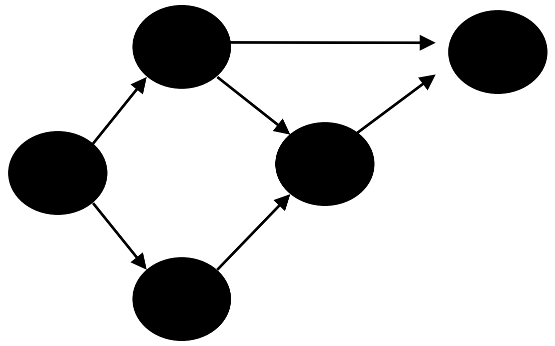
- The tree is simply drawn differently and with undirected edges



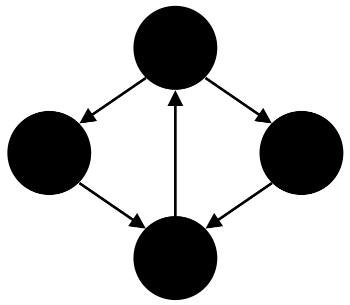
Directed Acyclic Graphs (DAGs)

A DAG is a directed graph with no directed cycles

- Every rooted directed tree is a DAG
- But not every DAG is a rooted directed tree



- Every DAG is a directed graph
- But not every directed graph is a DAG



Examples Again

Which of our directed-graph examples do you expect to be a DAG?

- Web pages with links
- Facebook friends
- "Input data" for the Kevin Bacon game
- Methods in a program that call each other
- Road maps
- Airline routes
- Family trees
- Course pre-requisites

Density / Sparsity

Recall:

In an undirected graph, $0 \leq |E| < |V|^2$

Recall:

In a directed graph, $0 \leq |E| \leq |V|^2$

So for any graph, $|E|$ is $O(|V|^2)$

Another fact:

If an undirected graph is *connected*,
then $|E| \geq |V|-1$ (pigeonhole principle)

Density / Sparsity

$|\mathrm{E}|$ is often much smaller than its maximum size

We do not always approximate as $|\mathrm{E}|$ as $O(|\mathrm{V}|^2)$

- This is a correct bound, but often not tight

If $|\mathrm{E}|$ is $\Theta(|\mathrm{V}|^2)$ (the bound is tight), we say the graph is dense

- More sloppily, dense means "lots of edges"

If $|\mathrm{E}|$ is $O(|\mathrm{V}|)$ we say the graph is sparse

- More sloppily, sparse means "most possible edges missing"

Insert humorous statement here

GRAPH DATA STRUCTURES

What's the Data Structure?

Graphs are often useful for lots of data and questions

- Example: "What's the lowest-cost path from x to y"

But we need a data structure that represents graphs

Which data structure is "best" can depend on:

- properties of the graph (e.g., dense versus sparse)
- the common queries about the graph ("is (u, v) an edge?" vs "what are the neighbors of node u?")

We will discuss two standard graph representations

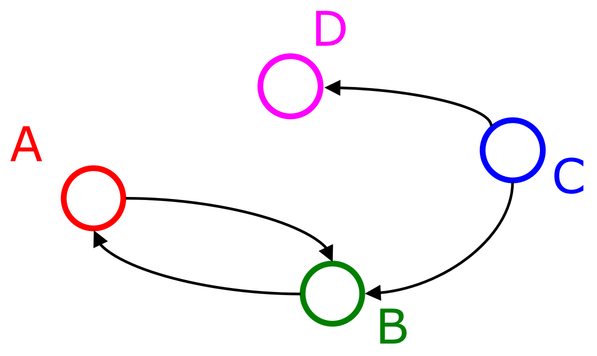
- Adjacency Matrix and Adjacency List
- Different trade-offs, particularly time versus space

Adjacency Matrix

Assign each node a number from 0 to $|V|-1$

A |V| x |V| matrix of Booleans (or 0 vs. 1)

- Then $M[u][v] == \text{true}$ means there is an edge from u to v



	A	B	C	D
A	F	T	F	F
B	T	F	F	F
C	F	T	F	T
D	F	F	F	F

Adjacency Matrix Properties

Running time to:

- Get a vertex's out-edges:
- Get a vertex's in-edges:
- Decide if some edge exists:
- Insert an edge:
- Delete an edge:

Space requirements:

Best for sparse or dense graphs?

	A	B	C	D
A	F	T	F	F
B	T	F	F	F
C	F	T	F	T
D	F	F	F	F

Adjacency Matrix Properties

Running time to:

- Get a vertex's out-edges: $O(|V|)$
- Get a vertex's in-edges: $O(|V|)$
- Decide if some edge exists: $O(1)$
- Insert an edge: $O(1)$
- Delete an edge: $O(1)$

$$\begin{array}{c|c|c|c|c|}
 & A & B & C & D \\
\hline
A & F & T & F & F \\
\hline
B & T & F & F & F \\
\hline
C & F & T & F & T \\
\hline
D & F & F & F & F \\
\hline
\end{array}$$

Space requirements:

$O(|V|^2)$

Best for sparse or dense graphs? *dense*

Adjacency Matrix Properties

How will the adjacency matrix vary for an undirected graph?

- Will be symmetric about diagonal axis
- Matrix: Could we save space by using only about half the array?

	A	B	C	D
A	F	T	F	F
B	T	F	F	F
C	*F*	*T*	*F*	T
D	*F*	*F*	*T*	*F*

- But how would you "get all neighbors"?

Adjacency Matrix Properties

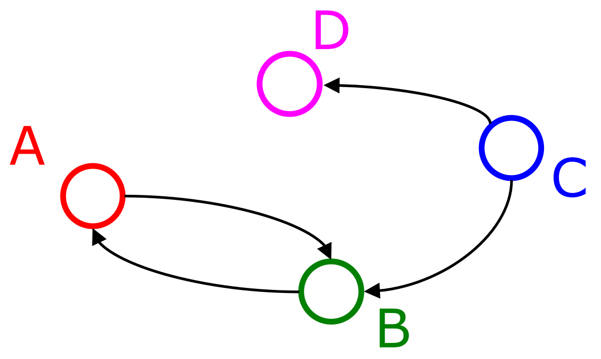
How can we adapt the representation for weighted graphs?

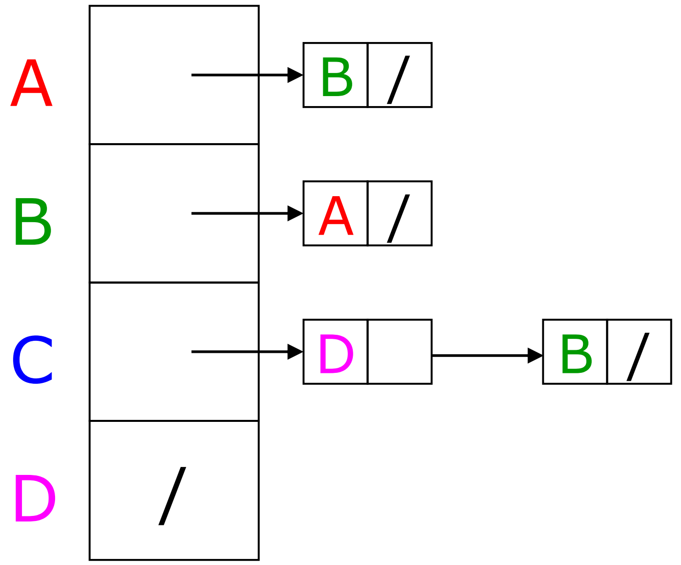
- Instead of Boolean, store a number in each cell
- Need some value to represent ‘not an edge’
  - 0, -1, or some other value based on how you are using the graph
  - Might need to be a separate field if no restrictions on weights

Adjacency List

Assign each node a number from 0 to $|V|$-1

- An array of length $|V|$ in which each entry stores a list of all adjacent vertices (e.g., linked list)

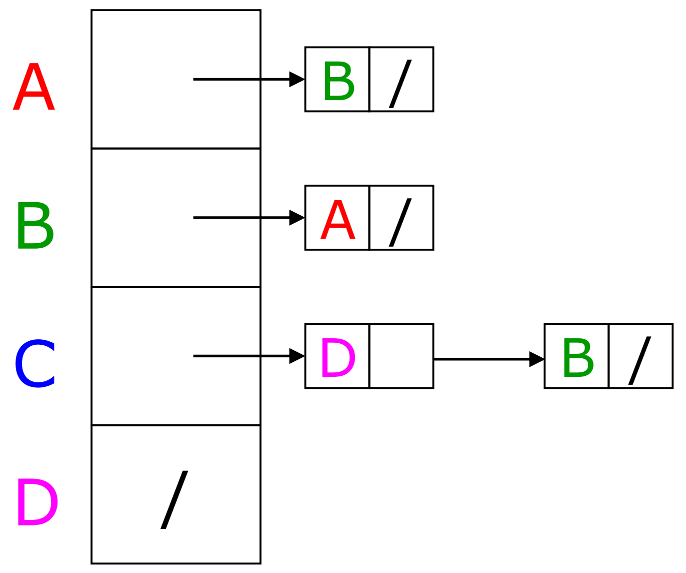




Adjacency List Properties

Running time to:

- Get a vertex's out-edges:
- Get a vertex's in-edges:
- Decide if some edge exists:
- Insert an edge:
- Delete an edge:



Space requirements:

Best for sparse or dense graphs?

Adjacency List Properties

Running time to:

- Get a vertex's out-edges:
  $O(d)$ where d is out-degree of vertex
- Get a vertex's in-edges:
  $O(|E|)$ (could keep a second adjacency list for this!)
- Decide if some edge exists:
  $O(d)$ where d is out-degree of source
- Insert an edge:
  $O(1)$ (unless you need to check if it's already there)
- Delete an edge:
  $O(d)$ where d is out-degree of source

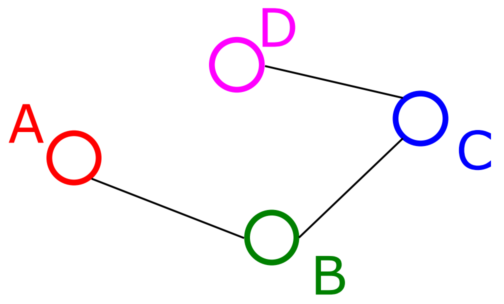
Space requirements: $O(|V|+|E|)$

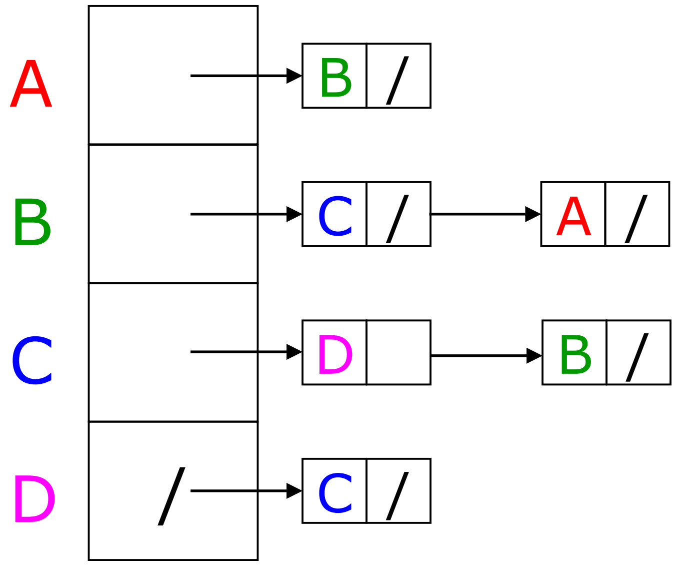
Best for sparse or dense graphs? *sparse*

Undirected Graphs

Adjacency lists also work well for undirected graphs with one caveat

- Put each edge in two lists to support efficient "get all neighbors"





Which is better?

Graphs are often sparse

- Streets form grids
- Airlines rarely fly to all cities

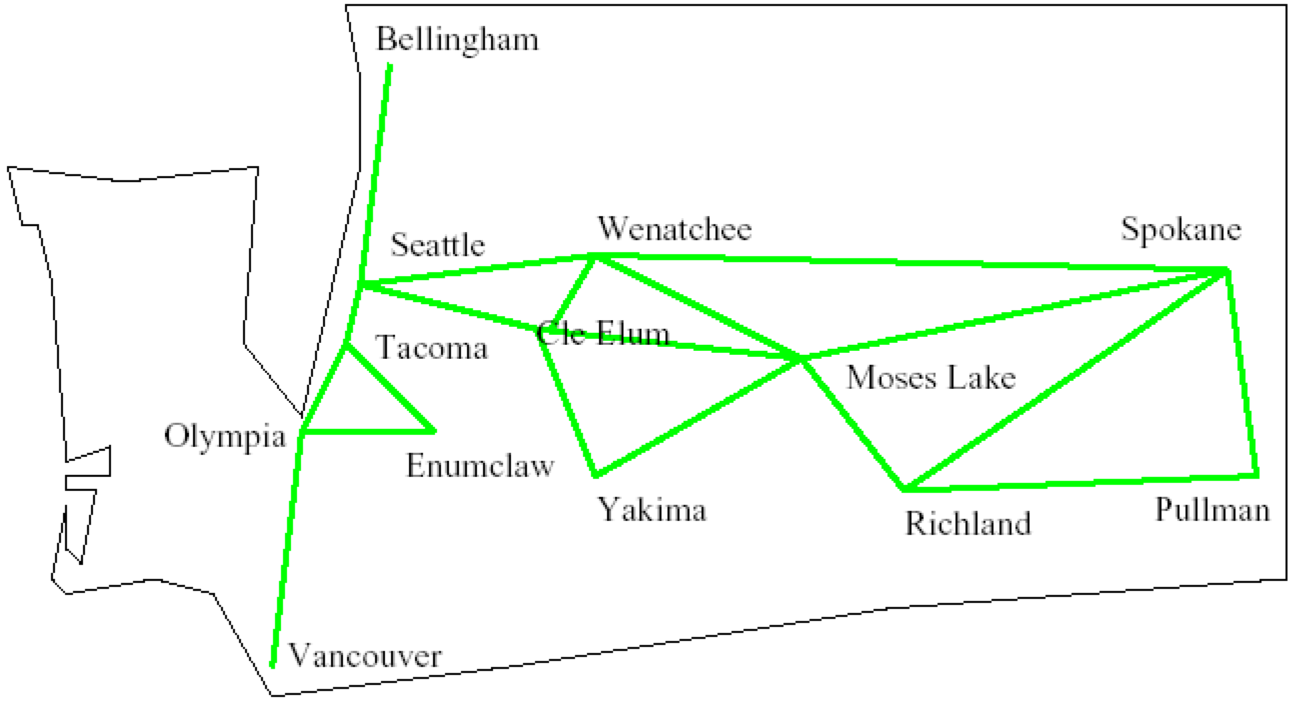
Adjacency lists should generally be your default choice

- Slower performance compensated by greater space savings

Might be easier to list what isn't a graph application…

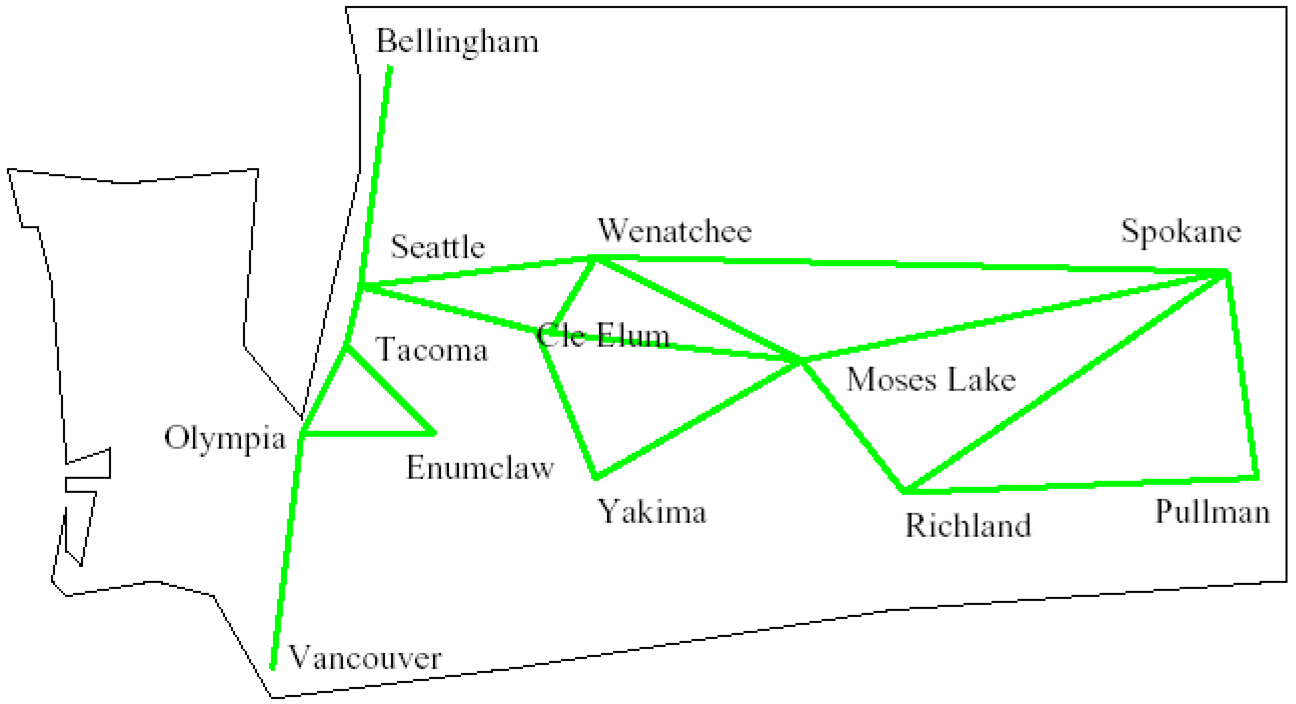
APPLICATIONS OF GRAPHS: TRAVERSALS

Application: Moving Around WA State



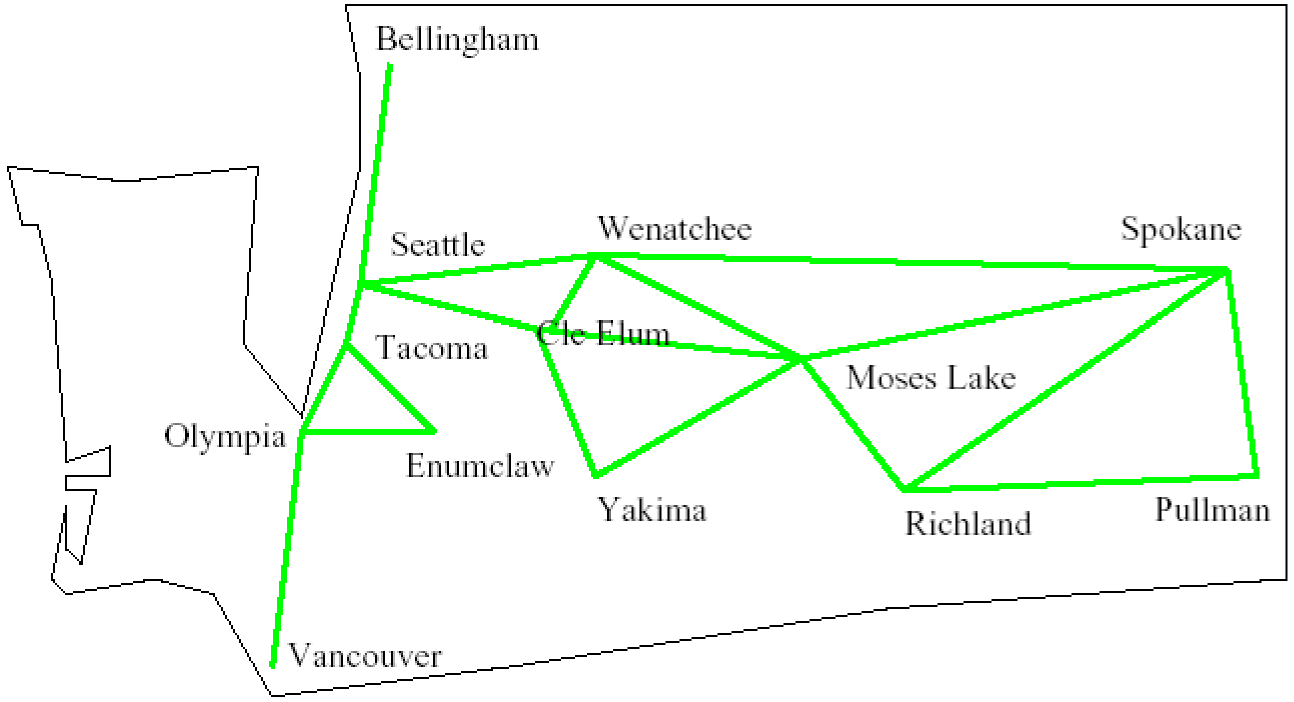
What's the *shortest way* to get from Seattle to Pullman?

Application: Moving Around WA State



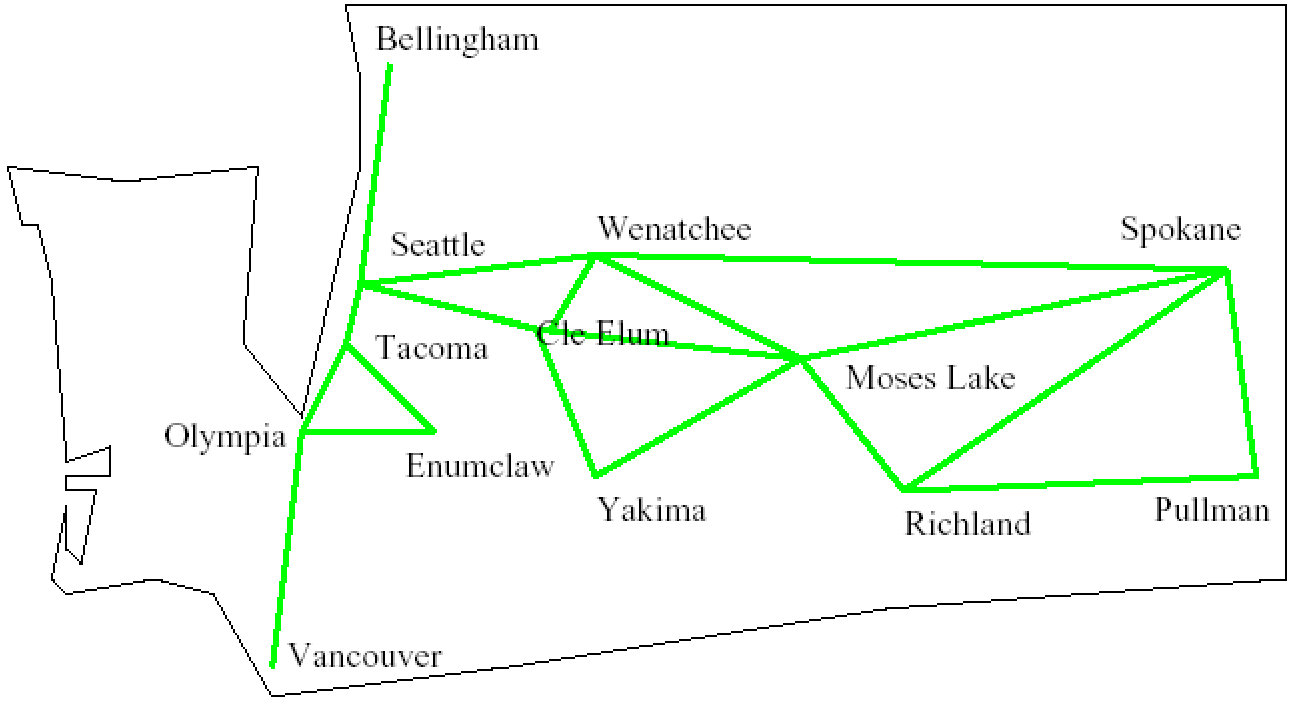
What's the *fastest way* to get from Seattle to Pullman?

Application: Reliability of Communication



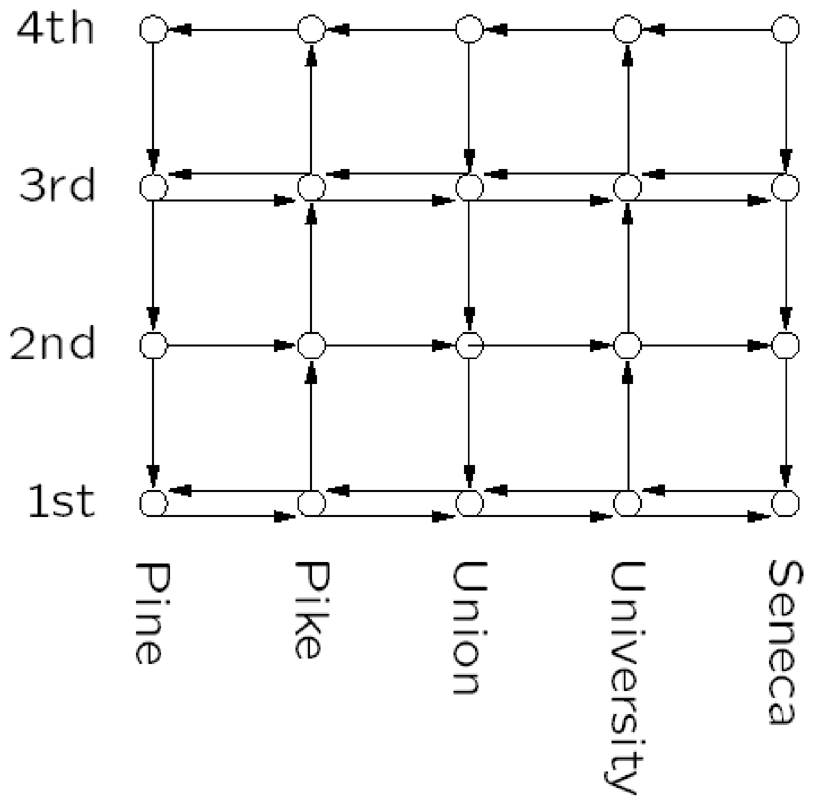
If Wenatchee's phone exchange *goes down*,
can Seattle still talk to Pullman?

Application: Reliability of Communication



If Tacomas's phone exchange *goes down*, can Olympia still talk to Spokane?

Applications: Bus Routes Downtown



If we’re at 3rd and Pine, how can we get to 1st and University using Metro?
How about 4th and Seneca?

Graph Traversals

For an arbitrary graph and a starting node v, find all nodes reachable from v (i.e., there exists a path)

- Possibly "do something" for each node (print to output, set some field, return from iterator, etc.)

Related Problems:

- Is an undirected graph connected?
- Is a digraph weakly/strongly connected?
  - For strongly, need a cycle back to starting node

Graph Traversals

Basic Algorithm for Traversals:

- Select a starting node
- Make a set of nodes adjacent to current node
- Visit each node in the set but "mark" each nodes after visiting them so you don't revisit them (and eventually stop)
- Repeat above but skip "marked nodes"

In Rough Code Form

```
traverseGraph(Node start) {
    Set pending = emptySet();
    pending.add(start)
    mark start as visited
    while(pending is not empty) {
        next = pending.remove()
        for each node u adjacent to next
            if(u is not marked) {
                mark u
                pending.add(u)
            }
        }
    }
}
```

Running Time and Options

Assuming add and remove are $O(1)$, entire traversal is $O(|E|)$ if using an adjacency list

The order we traverse depends entirely on how add and remove work/are implemented

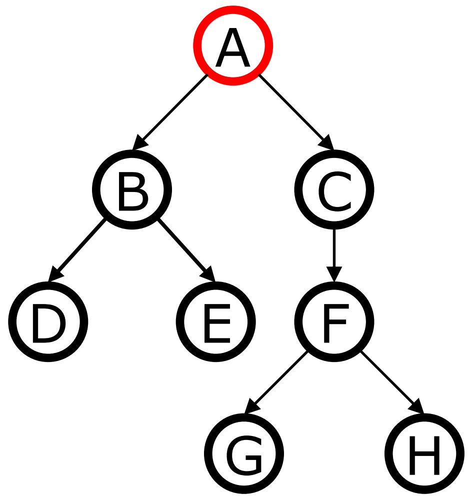
- DFS: a stack "depth-first graph search"
- BFS: a queue "breadth-first graph search"

DFS and BFS are "big ideas" in computer science

- Depth: recursively explore one part before going back to the other parts not yet explored
- Breadth: Explore areas closer to start node first

Recursive DFS, Example with Tree

A tree is a graph and DFS and BFS are particularly easy to "see" in one

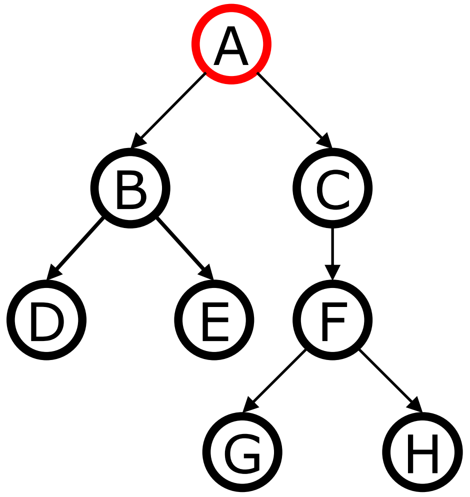


```
DFS(Node start) {
    mark and process start
    for each node u adjacent to start
        if u is not marked
            DFS(u)
}
```

Order processed: A, B, D, E, C, F, G, H

- This is a "pre-order traversal" for trees
- The marking is unneeded here but because we support arbitrary graphs, we need a means to process each node exactly once

DFS with Stack, Example with Tree

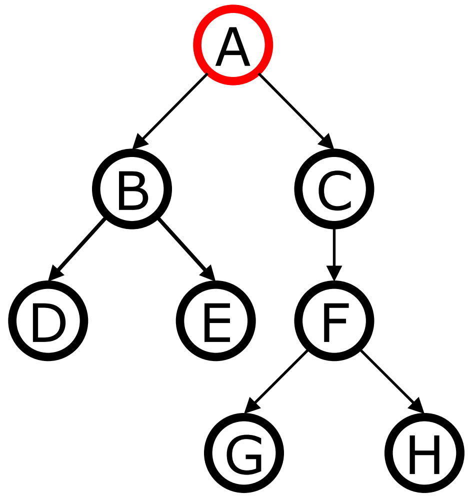


```
DFS2(Node start) {
   initialize stack s to hold start
   mark start as visited
   while(s is not empty) {
       next = s.pop() // and "process"
       for each node u adjacent to next
           if(u is not marked)
              mark u and push onto s
   }
}
```

Order processed: A, C, F, H, G, B, E, D

- A different order but still a perfectly fine traversal of the graph

BFS with Queue, Example with Tree



```
BFS(Node start) {
   initialize queue q to hold start
   mark start as visited
   while(q is not empty) {
      next = q.dequeue() // and "process"
      for each node u adjacent to next
         if(u is not marked)
            mark u and enqueue onto q
   }
}
```

Order processed: A, B, C, D, E, F, G, H

- A "level-order" traversal

DFS/BFS Comparison

BFS always finds the shortest path (or "optimal solution") from the starting node to a target node

- Storage for BFS can be extremely large
- A k-nary tree of height h could result in a queue size of k^h

DFS can use less space in finding a path

- If longest path in the graph is p and highest out-degree is d then DFS stack never has more than $d \cdot p$ elements

Implications

For large graphs, DFS is hugely more memory efficient, *if we can limit the maximum path length to some fixed d.*

If we *knew* the distance from the start to the goal in advance, we could simply *not add any children to stack after level d*

But what if we don’t know *d* in advance?

Iterative Deepening (IDFS)

Algorithms

- Try DFS up to recursion of K levels deep.
- If fails, increment K and start the entire search over

Performance:

- Like BFS, IDFS finds shortest paths
- Like DFS, IDFS uses less space
- Some work is repeated but minor compared to space savings

Saving the Path

Our graph traversals can answer the standard *reachability* question:

"Is there a path from node x to node y?"

But what if we want to actually output the path?

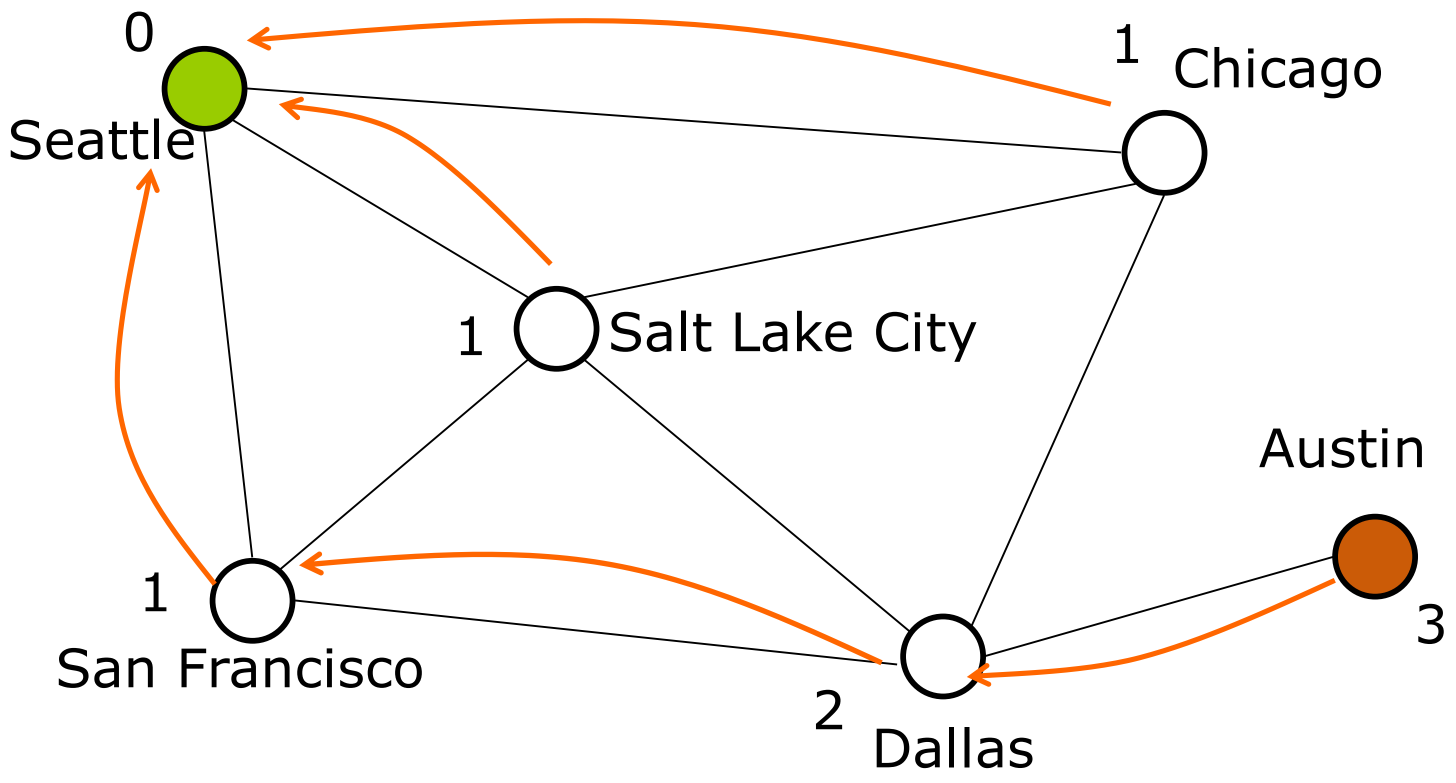
Easy:

- Store the previous node along the path:
  When processing *u* causes us to add *v* to the search, set *v.path* field to be *u*)
- When you reach the goal, follow path fields back to where you started (and then reverse the answer)
- What's an easy way to do the reversal? A Stack!!

Example using BFS

What is a path from Seattle to Austin?

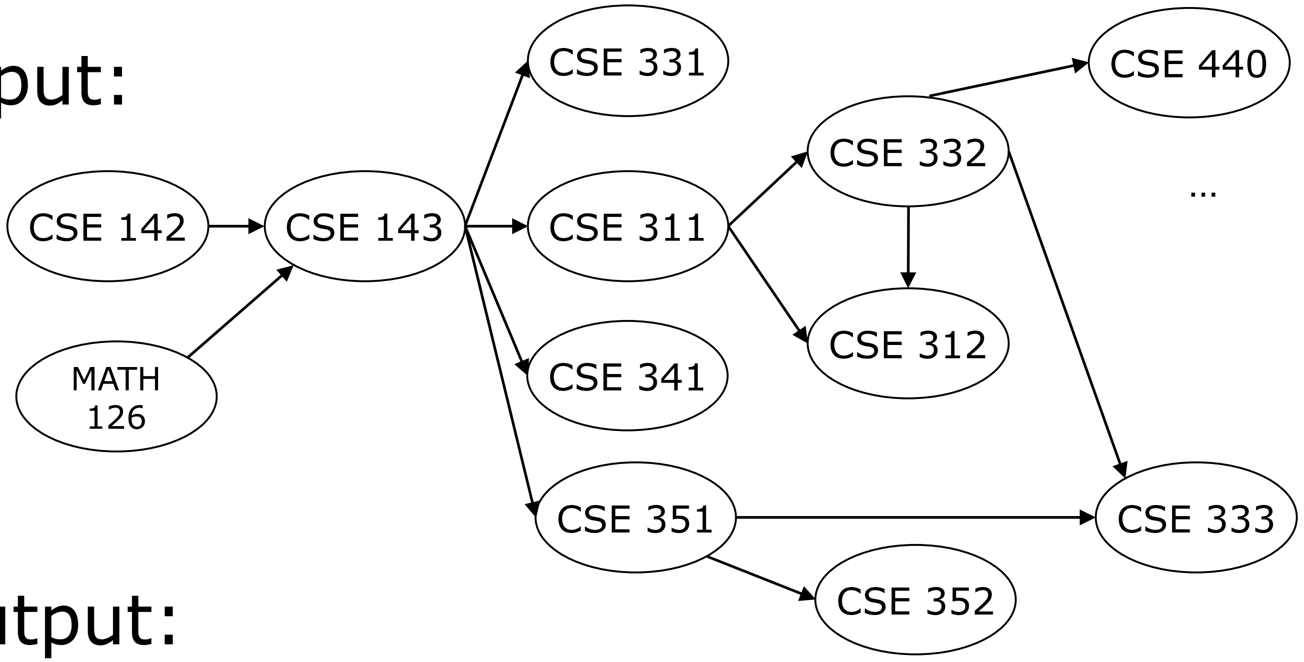
- Remember marked nodes are not re-enqueued
- Note shortest paths may not be unique



Topological Sort

Problem: Given a DAG $G=(V, E)$, output all the vertices in order such that if no vertex appears before any other vertex that has an edge to it

Example input:



Example output:

- 142, 126, 143, 311, 331, 332, 312, 341, 351, 333, 440, 352

Disclaimer: Do not use for official advising purposes!
(Implies that CSE 332 is a pre-req for CSE 312 – not true)

Questions and Comments

Terminology:

A DAG represents a partial order and a topological sort produces a total order that is consistent with it

Why do we perform topological sorts only on DAGs?

- Because a cycle means there is no correct answer

Is there always a unique answer?

- No, there can be one or more answers depending on the provided graph

What DAGs have exactly 1 answer?

- Lists

Uses Topological Sort

Figuring out how to finish your degree

Computing the order in which to recalculate cells in a spreadsheet

Determining the order to compile files with dependencies

In general, use a dependency graph to find an allowed order of execution

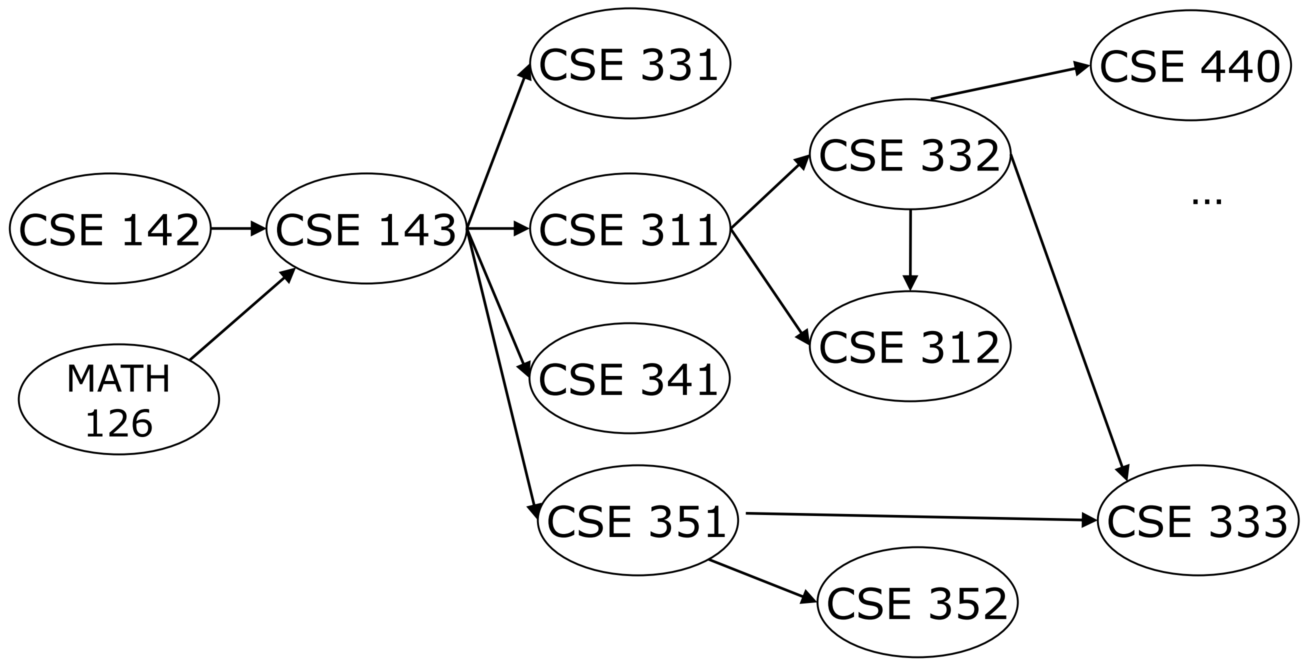
Topological Sort: First Approach

1. Label each vertex with its in-degree
   - Think "write in a field in the vertex"
   - You could also do this with a data structure on the side

2. While there are vertices not yet outputted:
   a) Choose a vertex **v** labeled with in-degree of 0
   b) Output **v** and "remove it" from the graph
   c) For each vertex **u** adjacent to **v**, decrement in-degree of **u**
      - (i.e., **u** such that (**v**,**u**) is in E)

Example

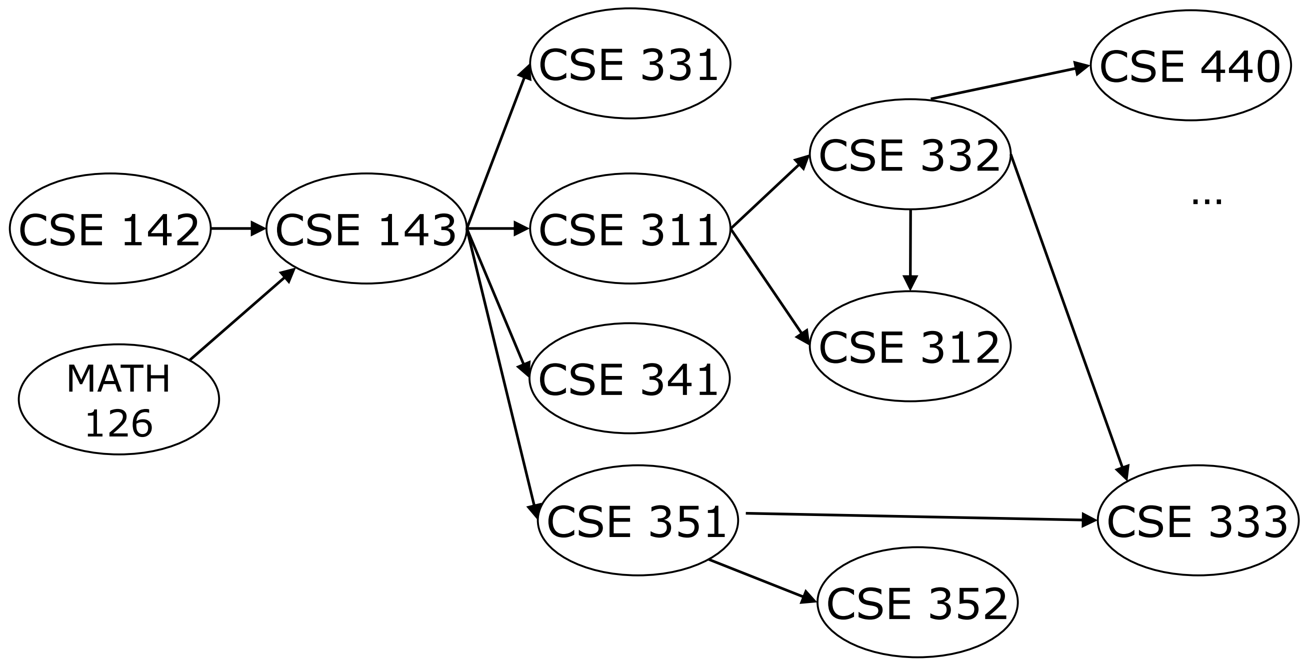
Output:



```
Node:    126 142 143 311 312 331 332 333 341 351 352 440
Removed?
In-deg:
```

Example

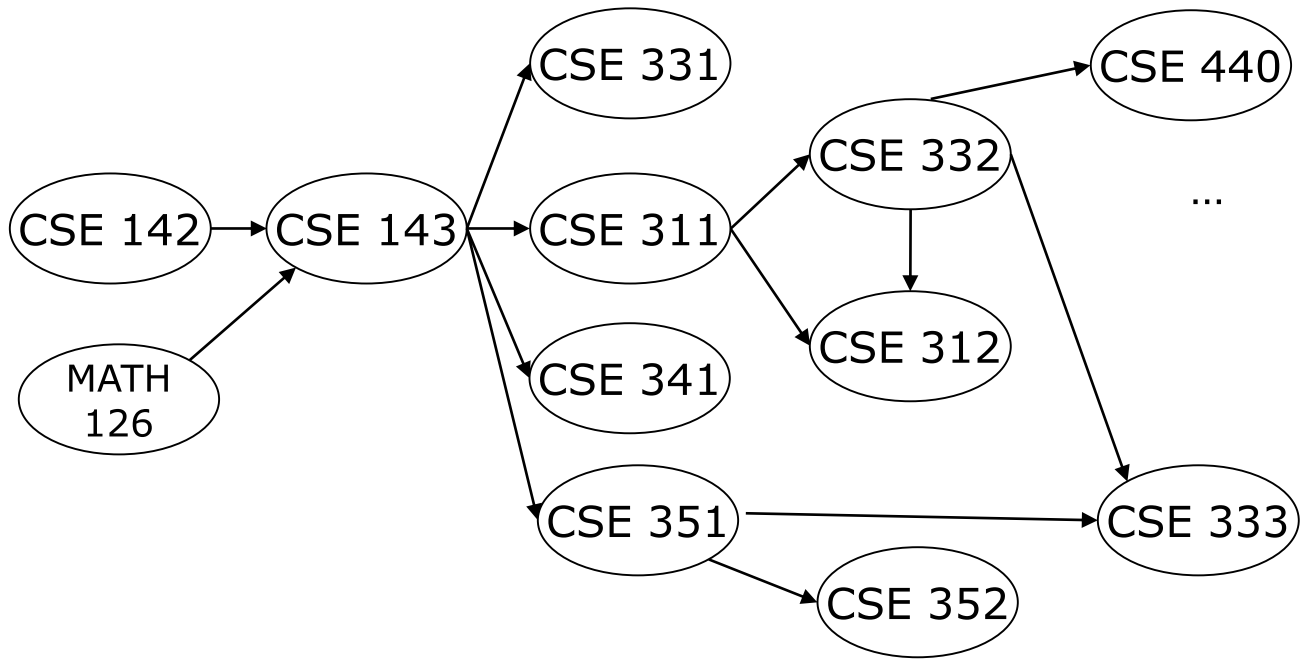
Output:



```
Node:     126 142 143 311 312 331 332 333 341 351 352 440
Removed?
In-deg:    0   0   2   1   2   1   1   2   1   1   1   1
```

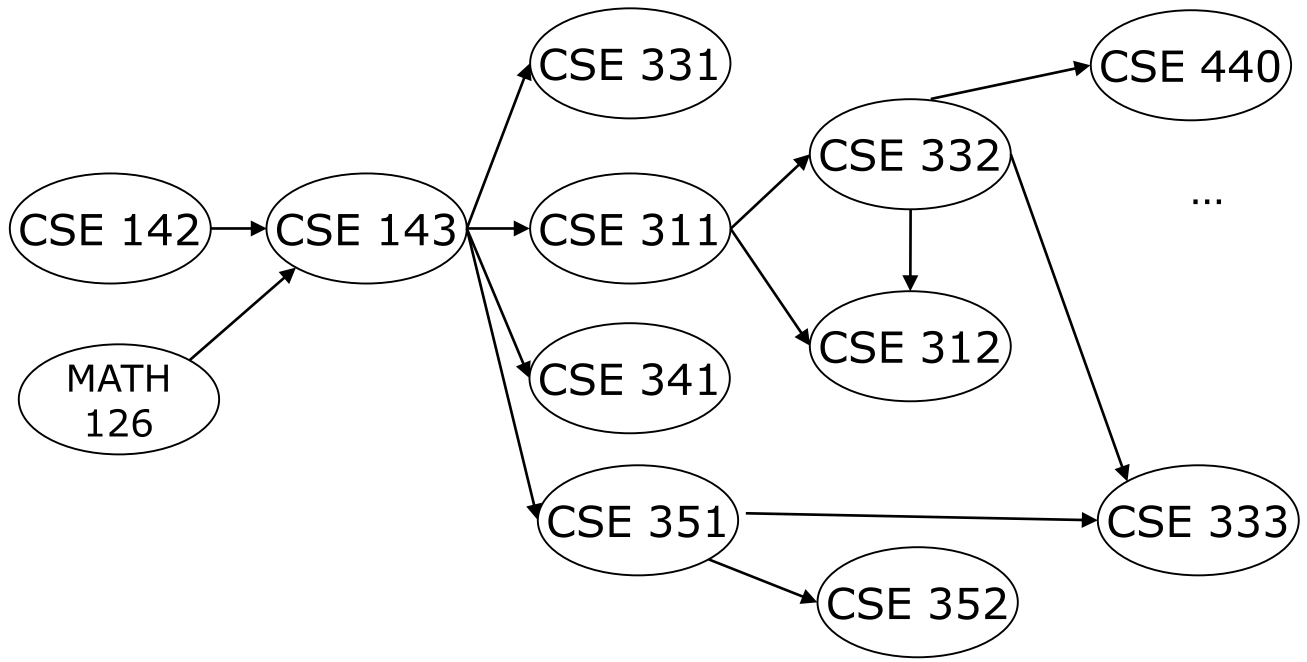
Example

Output:
126



Node:	126	142	143	311	312	331	332	333	341	351	352	440
Removed?	x											
In-deg:	0	0	2	1	2	1	1	2	1	1	1	1
			1									

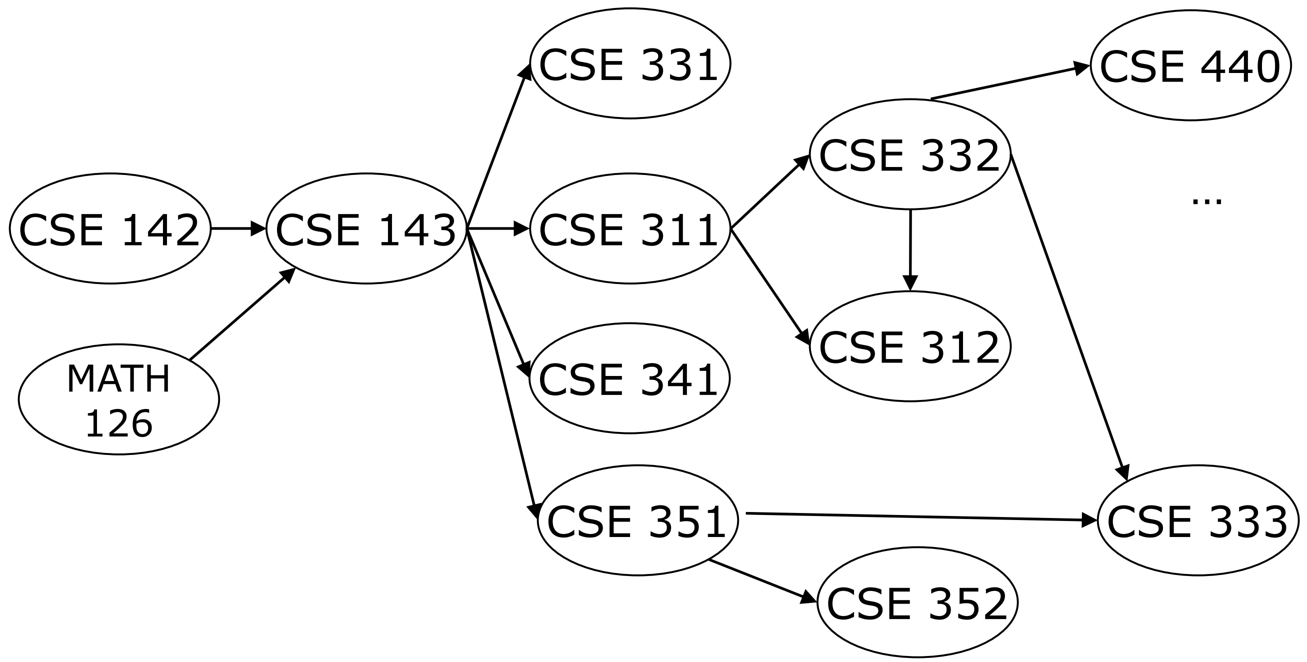
Example



Output:
126
142

```
Node:    126 142 143 311 312 331 332 333 341 351 352 440
Removed?  x   x
In-deg:   0   0   2   1   2   1   1   2   1   1   1   1
                  1
                  0
```

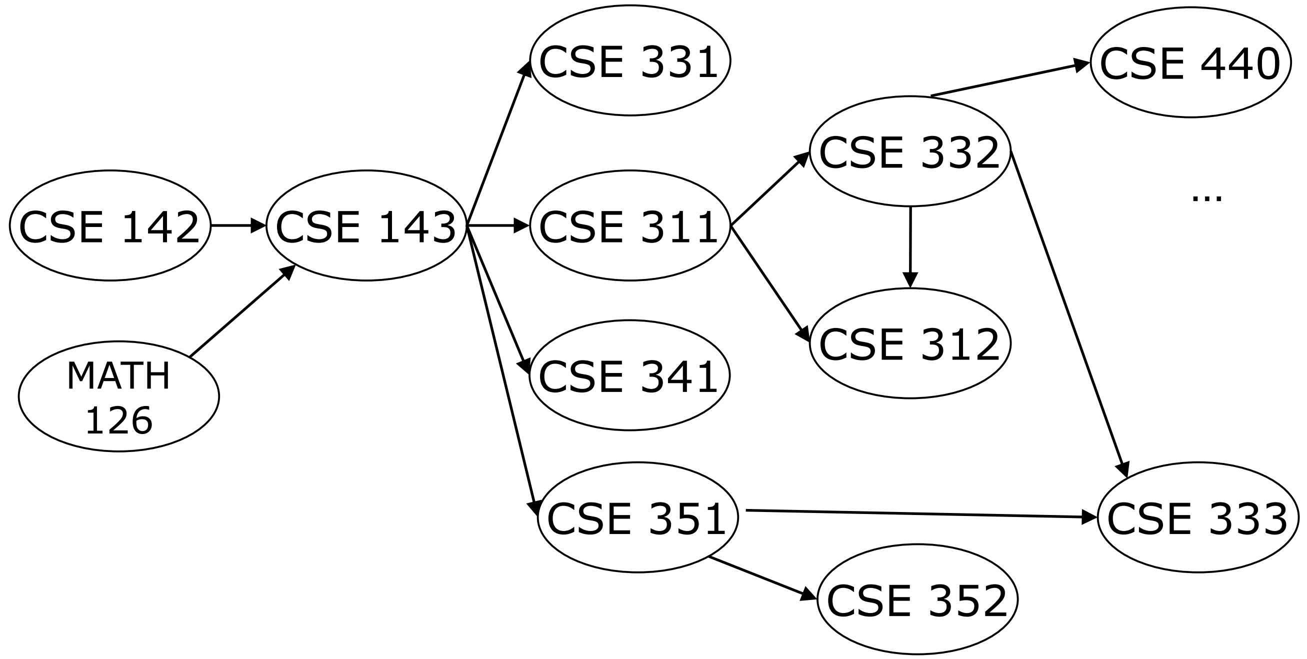
Example



Output:
126
142
143

Node:	126	142	143	311	312	331	332	333	341	351	352	440
Removed?	x	x	x									
In-deg:	0	0	2	1	2	1	1	2	1	1	1	1
			1	0		0			0	0		
			0									

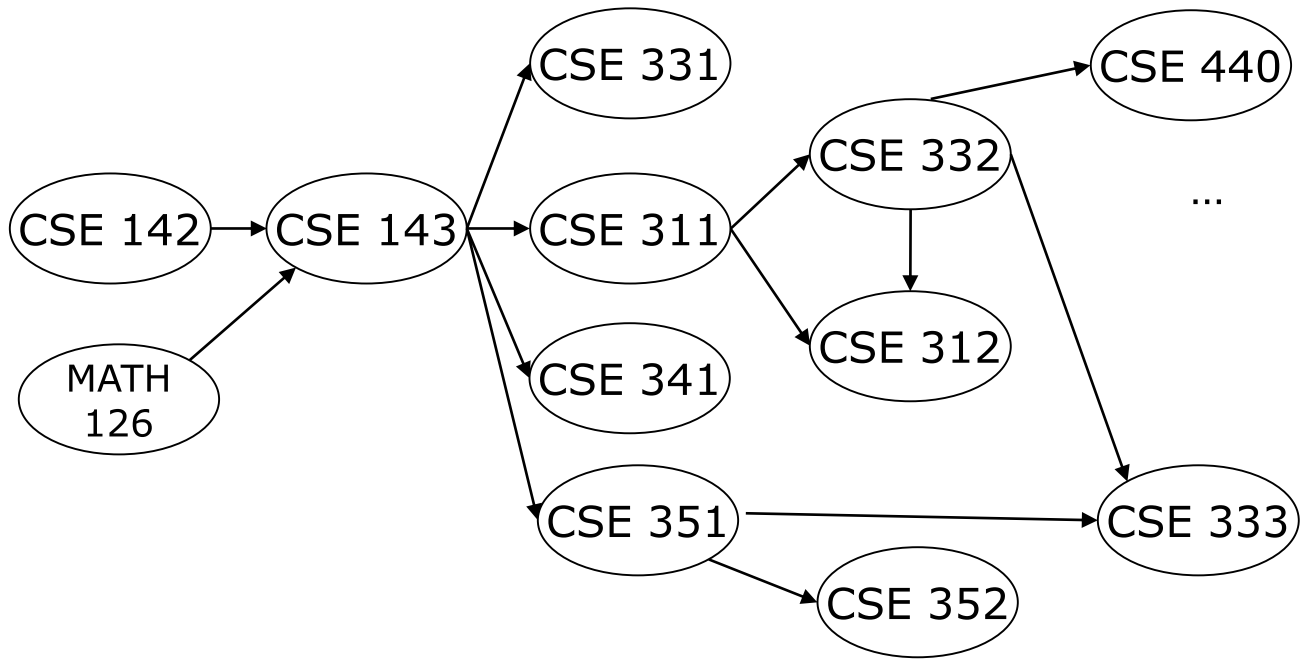
Example



Output:
126
142
143
311

Node:	126	142	143	311	312	331	332	333	341	351	352	440
Removed?	x	x	x	x								
In-deg:	0	0	2	1	2	1	1	2	1	1	1	1
			1	0	1	0	0		0	0		
			0									

Example

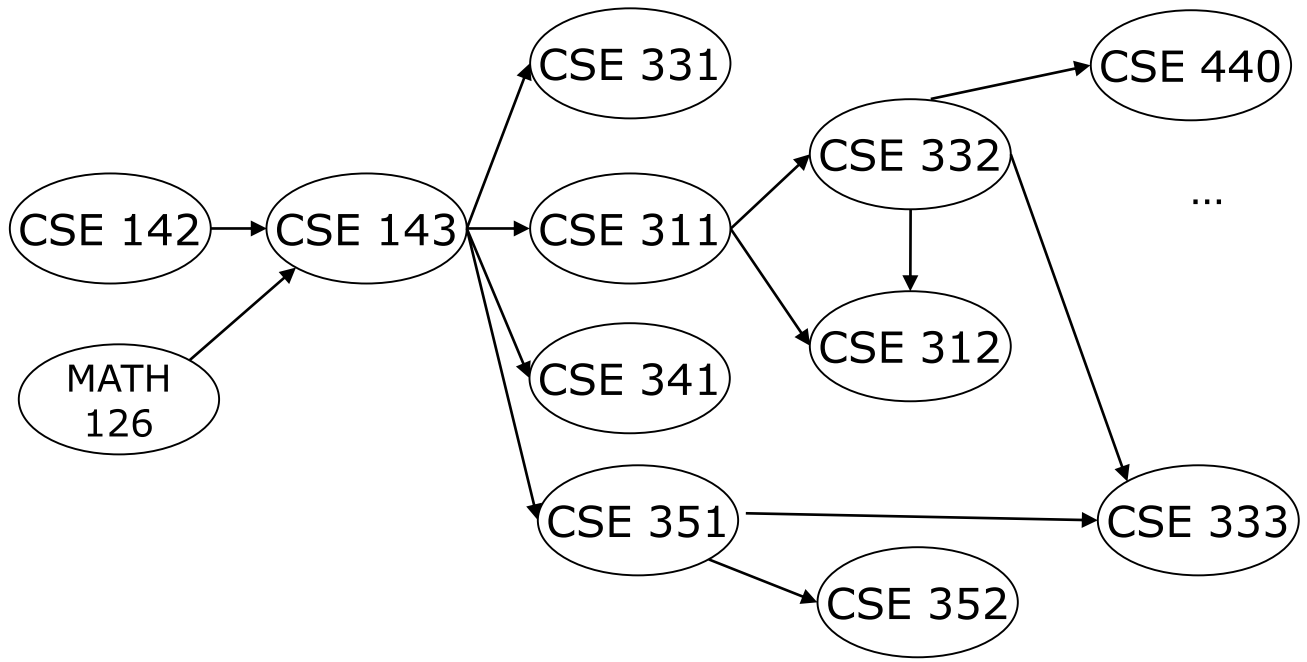


Output:

126
142
143
311
331

Node:	126	142	143	311	312	331	332	333	341	351	352	440
Removed?	x	x	x	x		x						
In-deg:	0	0	2	1	2	1	1	2	1	1	1	1
			1	0	1	0	0		0	0		
			0									

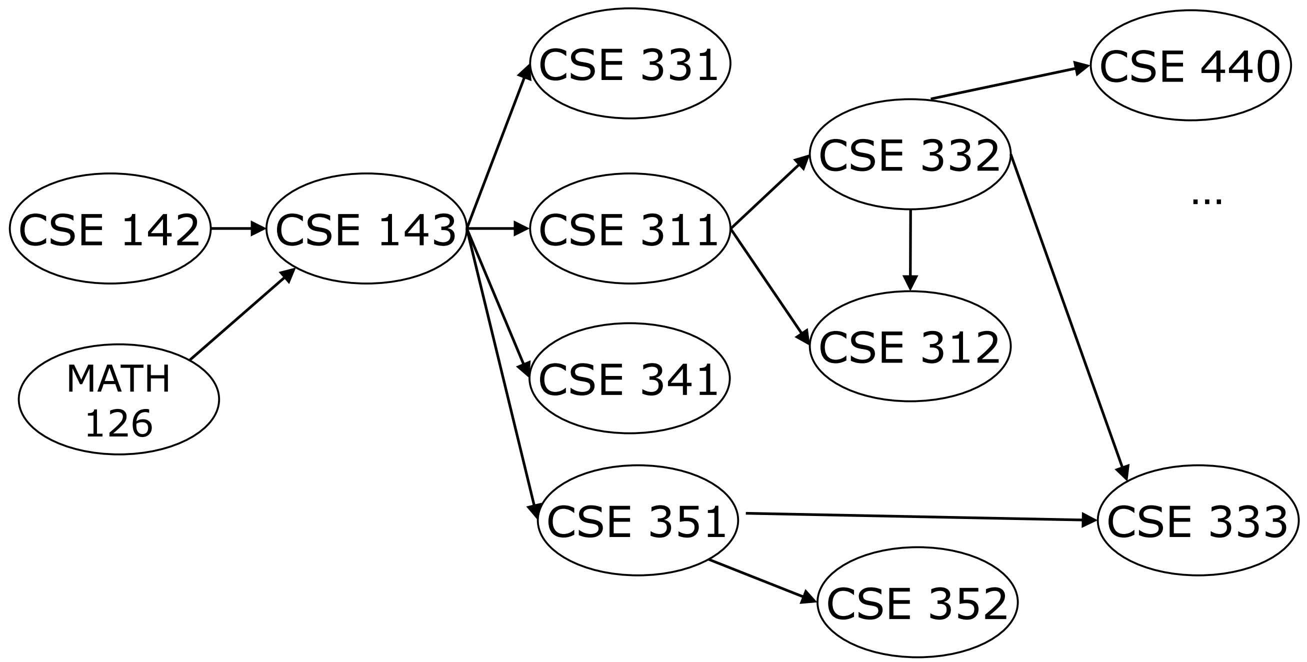
Example



Output:
126
142
143
311
331
332

Node:	126	142	143	311	312	331	332	333	341	351	352	440
Removed?	x	x	x	x		x	x					
In-deg:	0	0	2	1	2	1	1	2	1	1	1	1
			1	0	1	0	0	1	0	0		0
			0		0							

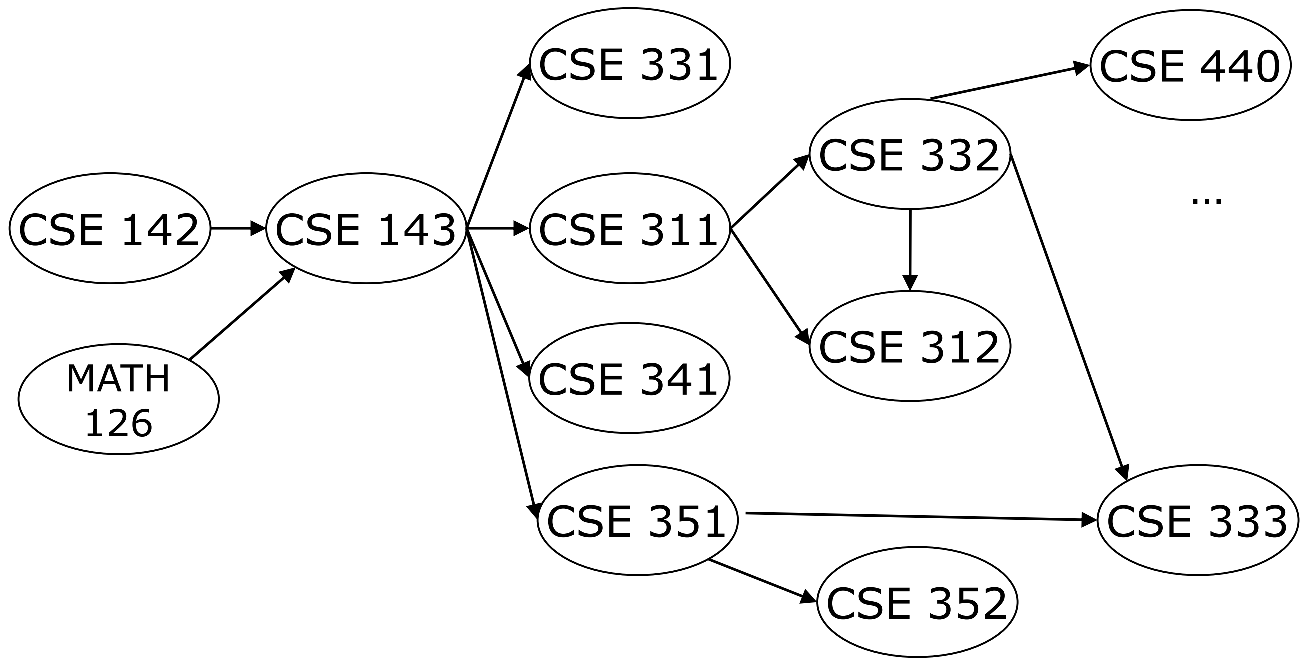
Example



Output:
126
142
143
311
331
332
312

Node:	126	142	143	311	312	331	332	333	341	351	352	440
Removed?	x	x	x	x	x	x	x					
In-deg:	0	0	2	1	2	1	1	2	1	1	1	1
			1	0	1	0	0	1	0	0		0
			0		0							

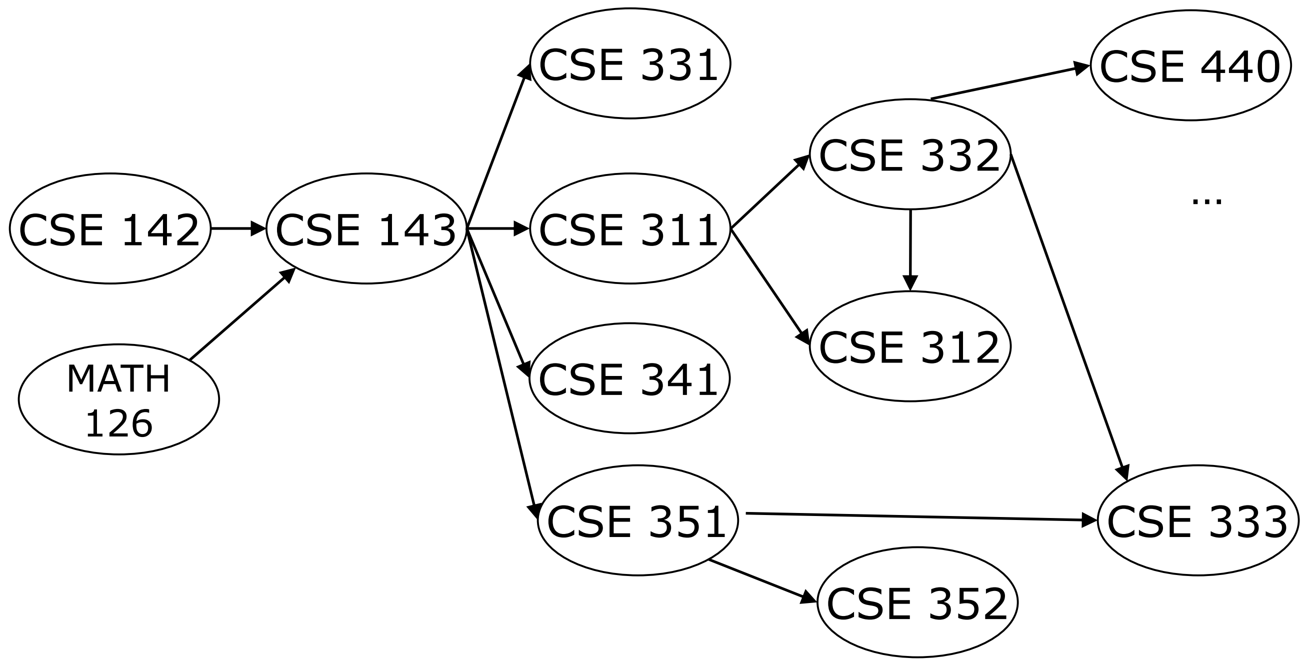
Example



Output:
126
142
143
311
331
332
312
341

Node:	126	142	143	311	312	331	332	333	341	351	352	440
Removed?	x	x	x	x	x	x	x		x			
In-deg:	0	0	2	1	2	1	1	2	1	1	1	1
			1	0	1	0	0	1	0	0		0
			0		0							

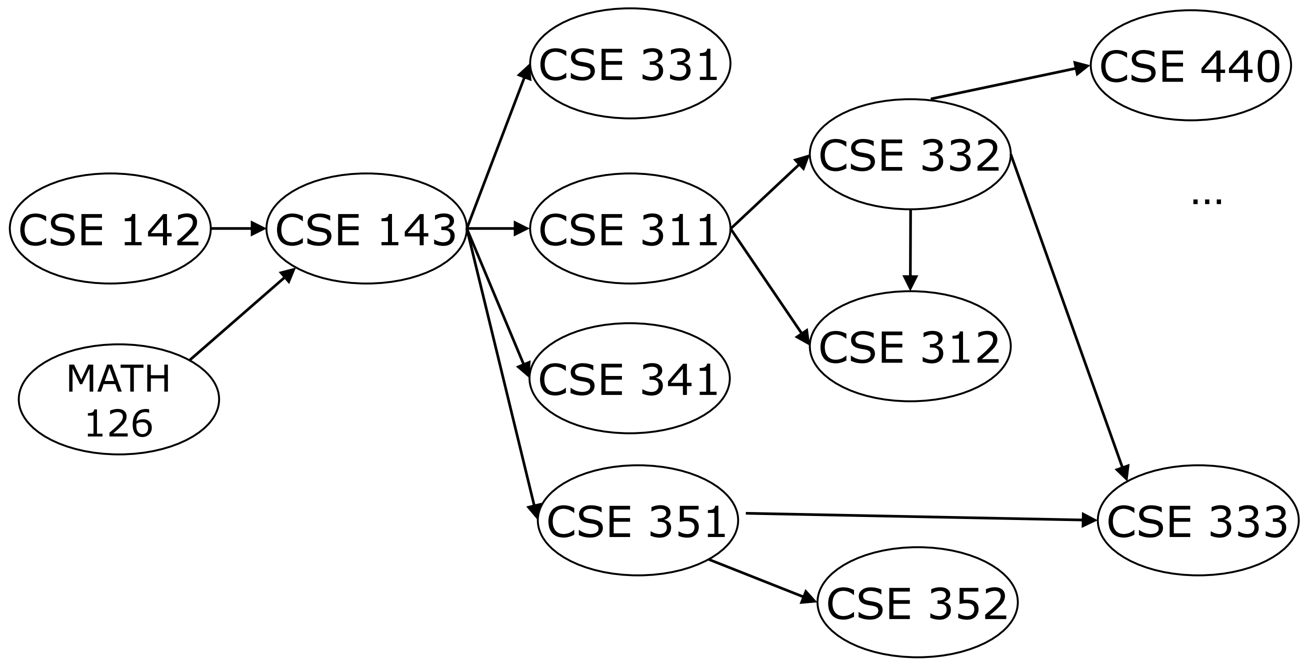
Example



Output:
126
142
143
311
331
332
312
341
351

Node:	126	142	143	311	312	331	332	333	341	351	352	440
Removed?	x	x	x	x	x	x	x		x	x		
In-deg:	0	0	2	1	2	1	1	2	1	1	1	1
			1	0	1	0	0	1	0	0	0	0
			0		0			0				

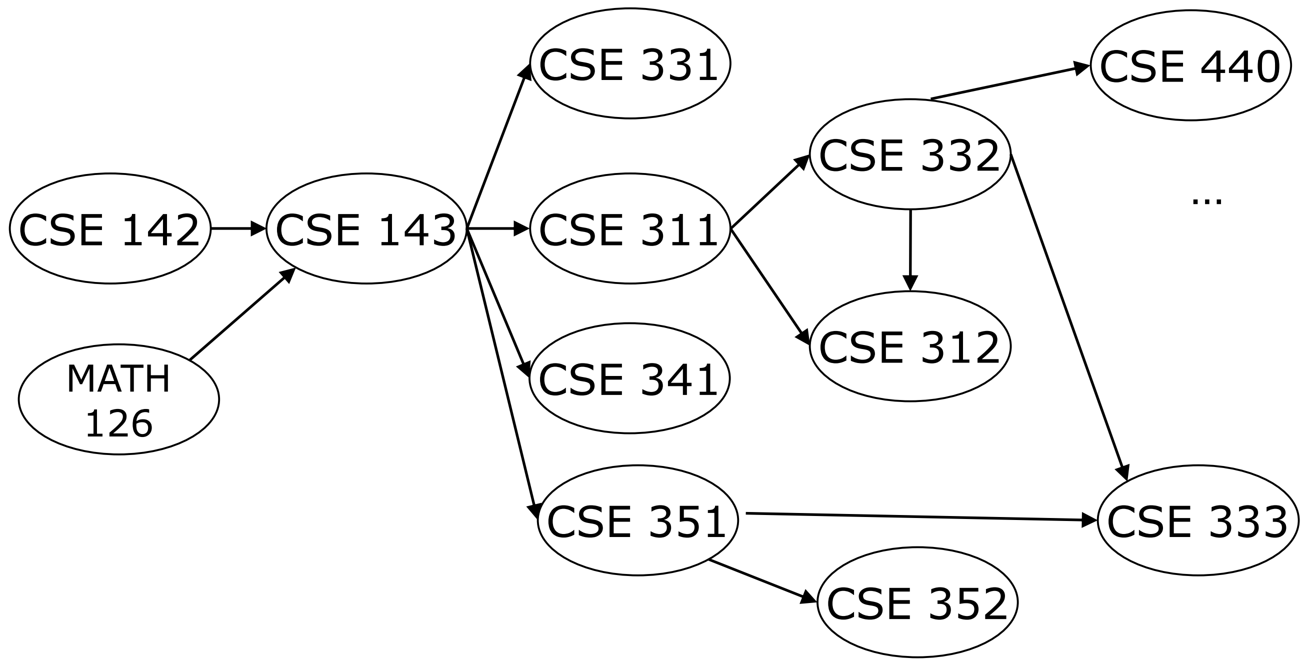
Example



Output:
126
142
143
311
331
332
312
341
351
333

Node:	126	142	143	311	312	331	332	333	341	351	352	440
Removed?	x	x	x	x	x	x	x	x	x	x		
In-deg:	0	0	2	1	2	1	1	2	1	1	1	1
			1	0	1	0	0	1	0	0	0	0
			0		0			0				

Example

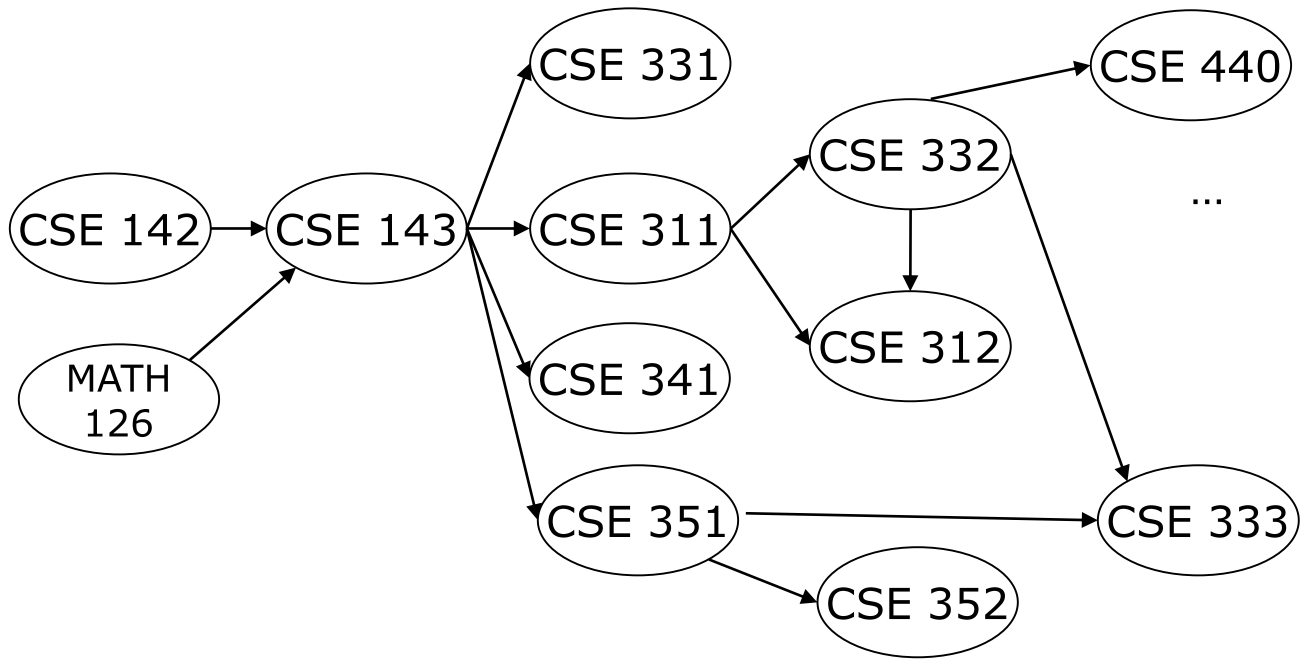


Output:

126 352
142
143
311
331
332
312
341
351
333

Node:	126	142	143	311	312	331	332	333	341	351	352	440
Removed?	x	x	x	x	x	x	x	x	x	x	x	
In-deg:	0	0	2	1	2	1	1	2	1	1	1	1
			1	0	1	0	0	1	0	0	0	0
			0		0			0				

Example



Output:

```
126  352
142  440
143
311
331
332
312
341
351
333
```

Node:	126	142	143	311	312	331	332	333	341	351	352	440
Removed?	x	x	x	x	x	x	x	x	x	x	x	x
In-deg:	0	0	2	1	2	1	1	2	1	1	1	1
			1	0	1	0	0	1	0	0	0	0
			0		0			0				

Running Time?

```
labelEachVertexWithItsInDegree();

for(i=0; i < numVertices; i++) {
      v = findNewVertexOfDegreeZero();
      put v next in output
      for each w adjacent to v
         w.indegree--;
}
```

What is the worst-case running time?

- Initialization $O(|V| + |E|)$ (assuming adjacency list)
- Sum of all find-new-vertex $O(|V|^2)$ (because each $O(|V|)$)
- Sum of all decrements $O(|E|)$ (assuming adjacency list)
- So total is $O(|V|^2 + |E|)$ – not good for a sparse graph!

Doing Better

Avoid searching for a zero-degree node every time!

- Keep the “pending” zero-degree nodes in a list, stack, queue, bag, or something that gives O(1) add/remove
- Order we process them affects the output but not correctness or efficiency

Using a queue:

- Label each vertex with its in-degree,
- Enqueue all 0-degree nodes
- While queue is not empty
  - v = dequeue()
  - Output v and remove it from the graph
  - For each vertex u adjacent to v, decrement the in-degree of u and if new degree is 0, enqueue it

Running Time?

```
labelAllWithIndegreesAndEnqueueZeros();

for(i=0; i < numVertices; i++) {
  v = dequeue();
  put v next in output
  for each w adjacent to v {
     w.indegree--;
     if(w.indegree==0)
       enqueue(w);
  }
}
```

- Initialization: $O(|V| + |E|)$ (assuming adjacency list)
- Sum of all enqueues and dequeues: $O(|V|)$
- Sum of all decrements: $O(|E|)$ (assuming adjacency list)
- So total is $O(|E| + |V|)$ – much better for sparse graph!

More Graph Algorithms

Finding a shortest path is one thing

- What happens when we consider weighted edges (as in distances)?

Next time we will discuss shortest path algorithms and the contributions of a curmudgeonly computer scientist

Unit- IV

Introduction to Algorithms

Introduction – Notion of Algorithm – Fundamentals of Algorithmic problem solving – Important problem types – Mathematical analysis for recursive & non recursive algorithms – Brute Fore – Selection Sort – Bubble Sort.

What is course about?

The theoretical study of design and analysis of computer algorithms

Basic goals for an algorithm:

- always correct
- always terminates
- This class: performance
- Performance often draws the line between what is possible and what is impossible.

Design and Analysis of Algorithms

- ***Analysis:*** predict the cost of an algorithm in terms of resources and performance

- ***Design:*** design algorithms which minimize the cost

The problem of sorting

Input: sequence $\langle a_1, a_2, \ldots, a_n \rangle$ of numbers.

Output: permutation $\langle a'_1, a'_2, \ldots, a'_n \rangle$ such that $a'_1 \leq a'_2 \leq \cdots \leq a'_n$.

Example:

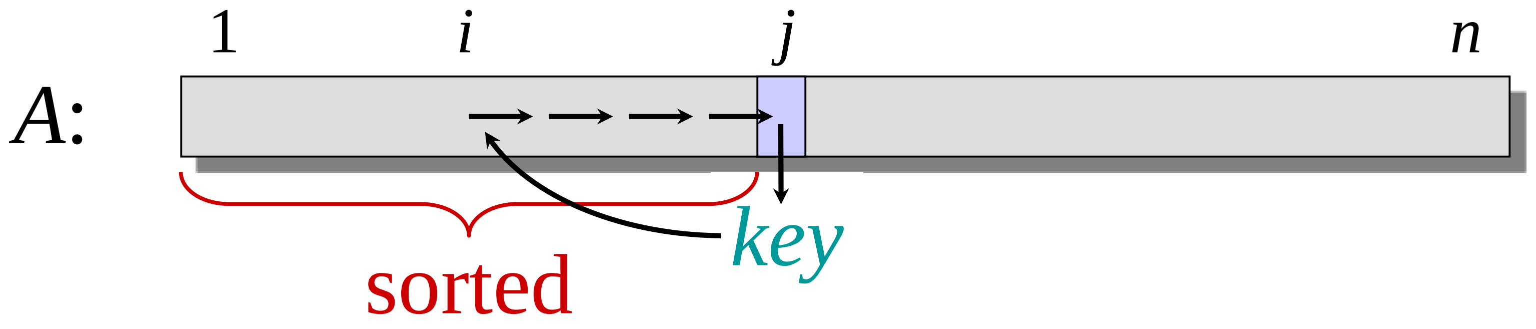
Input: 8 2 4 9 3 6

Output: 2 3 4 6 8 9

Insertion sort

“pseudocode”

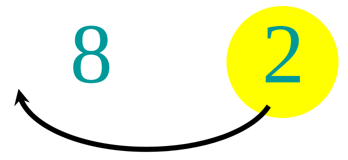
```
INSERTION-SORT $(A, n)$        ⊳ $A[1 \,..\, n]$
    for $j \leftarrow 2$ to $n$
        do $key \leftarrow A[j]$
           $i \leftarrow j - 1$
           while $i > 0$ and $A[i] > key$
               do $A[i+1] \leftarrow A[i]$
                  $i \leftarrow i - 1$
           $A[i+1] = key$
```



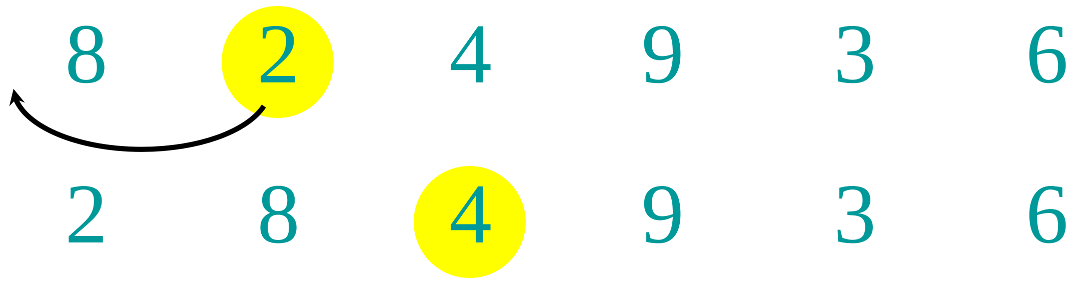
Example of insertion sort

8 $\quad$ 2 $\quad$ 4 $\quad$ 9 $\quad$ 3 $\quad$ 6

Example of insertion sort



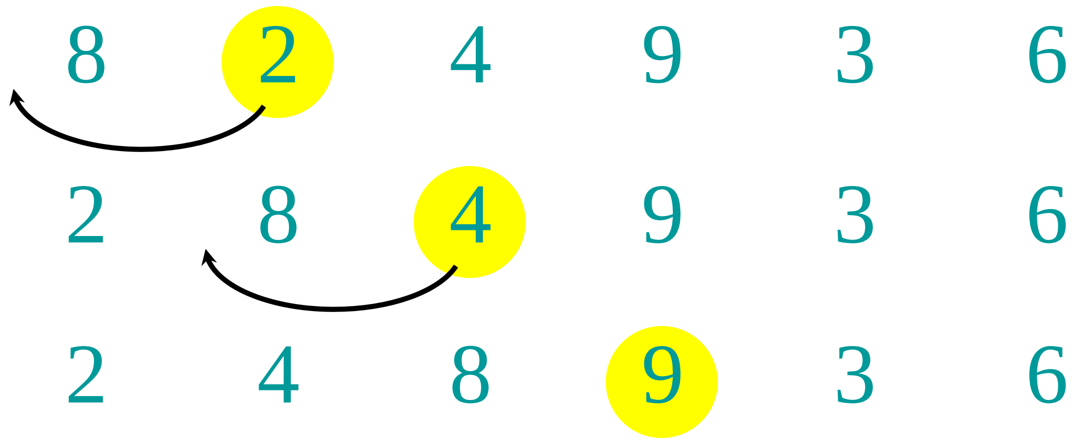
Example of insertion sort



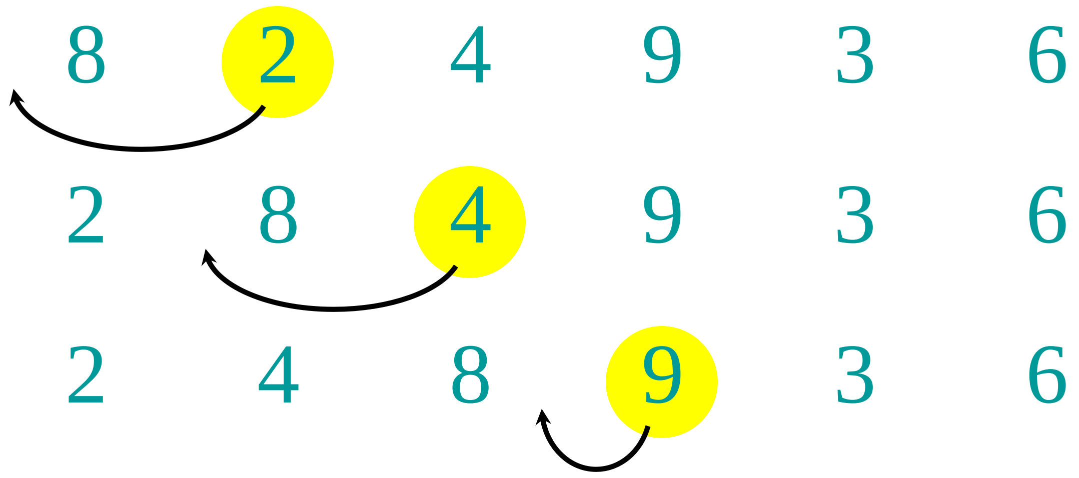
Example of insertion sort

$$\begin{tikzcd}
8 & 2 \arrow[l, bend left] & 4 & 9 & 3 & 6 \\
2 & 8 & 4 \arrow[l, bend left] & 9 & 3 & 6
\end{tikzcd}$$

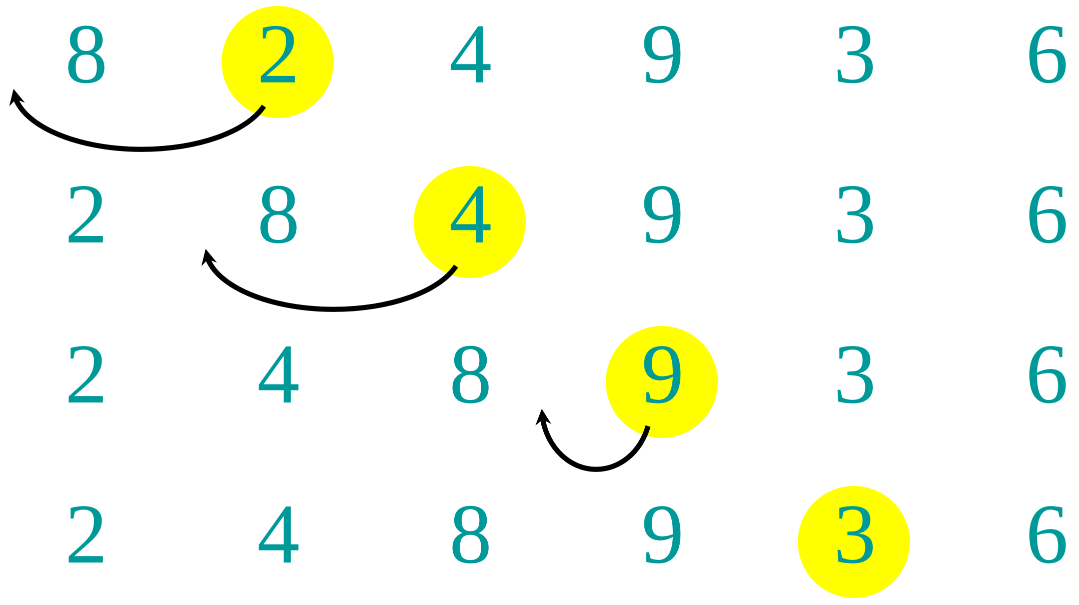
Example of insertion sort



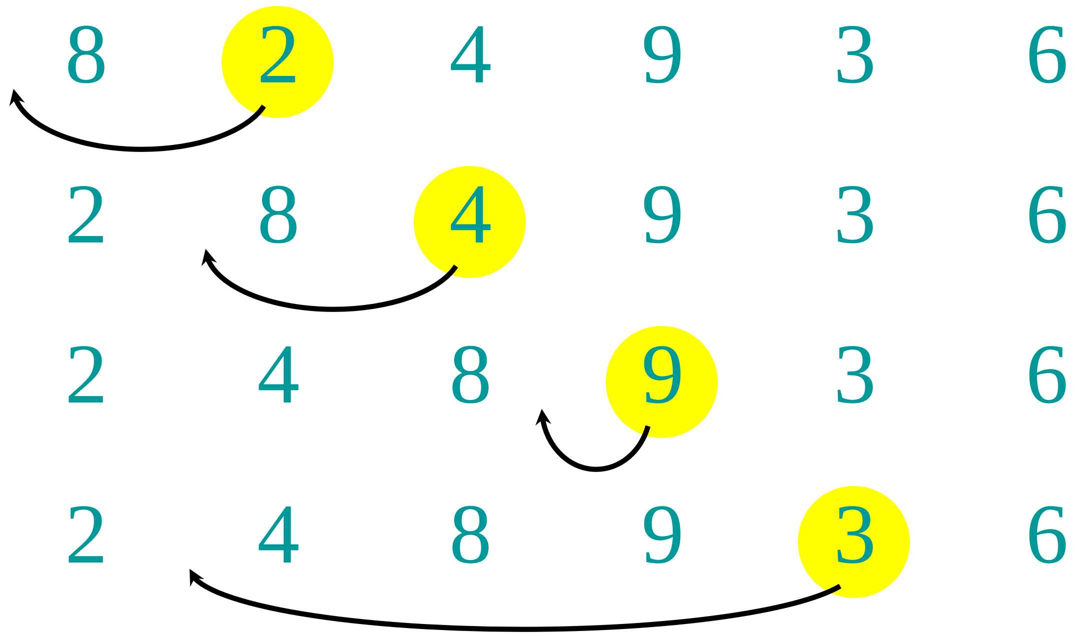
Example of insertion sort



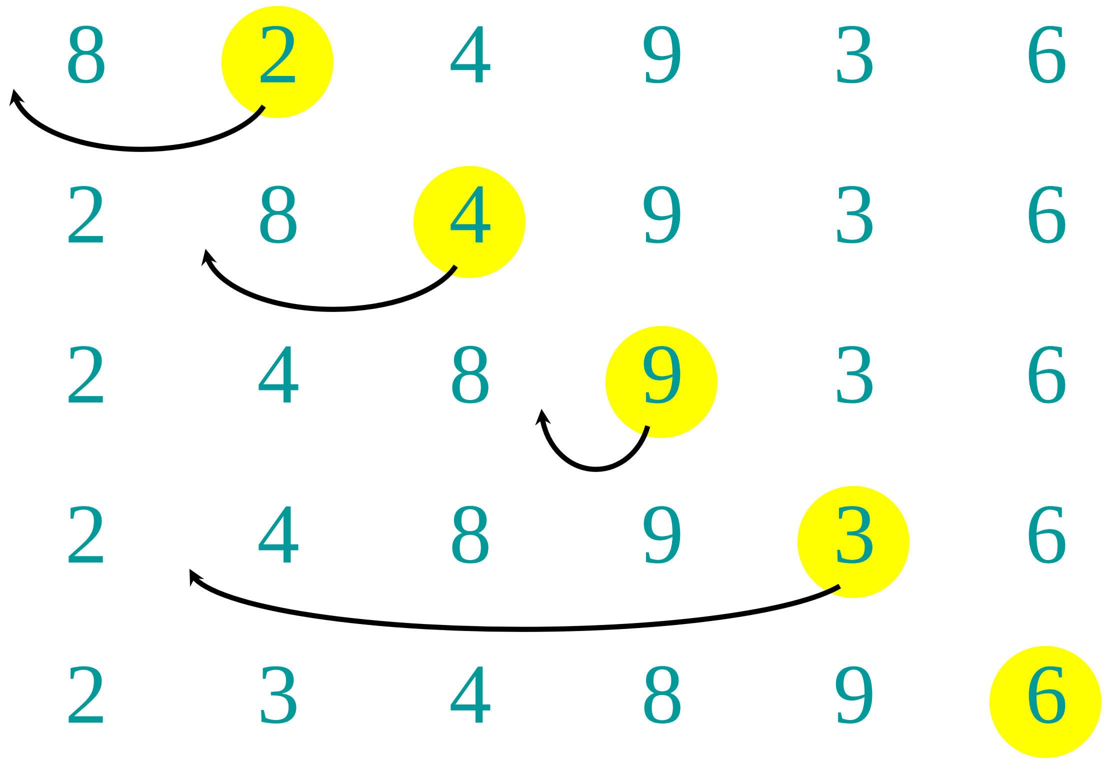
Example of insertion sort



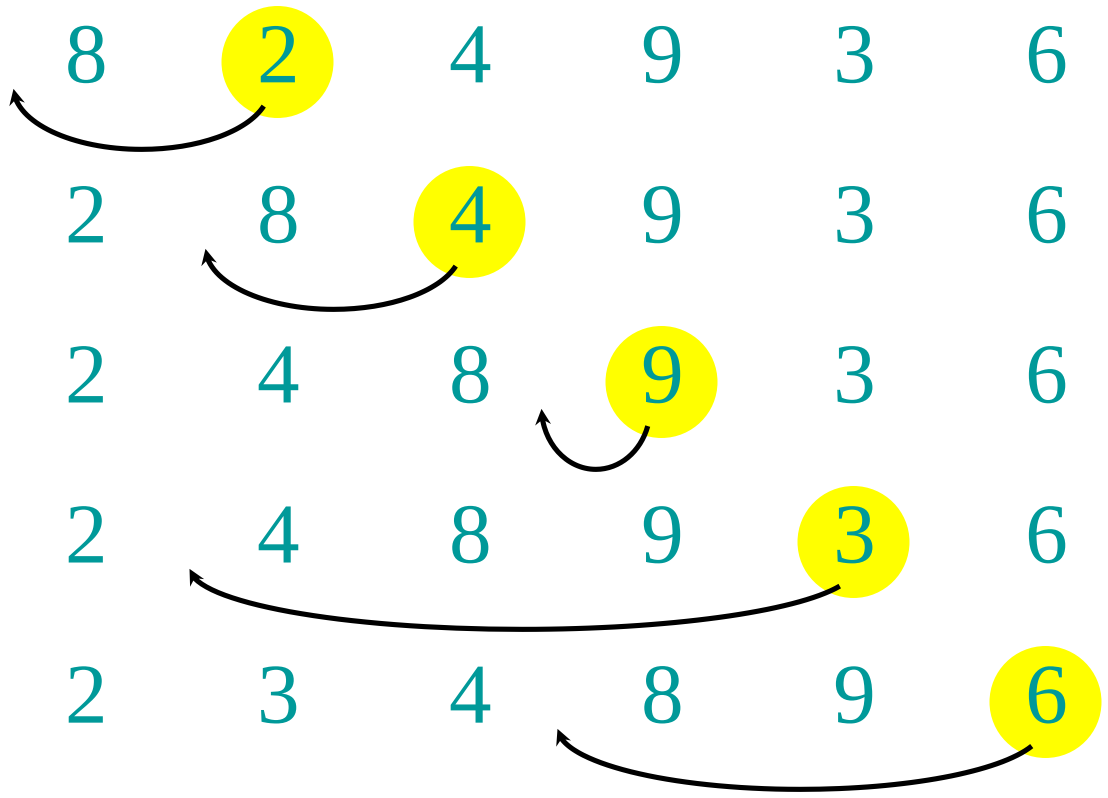
Example of insertion sort



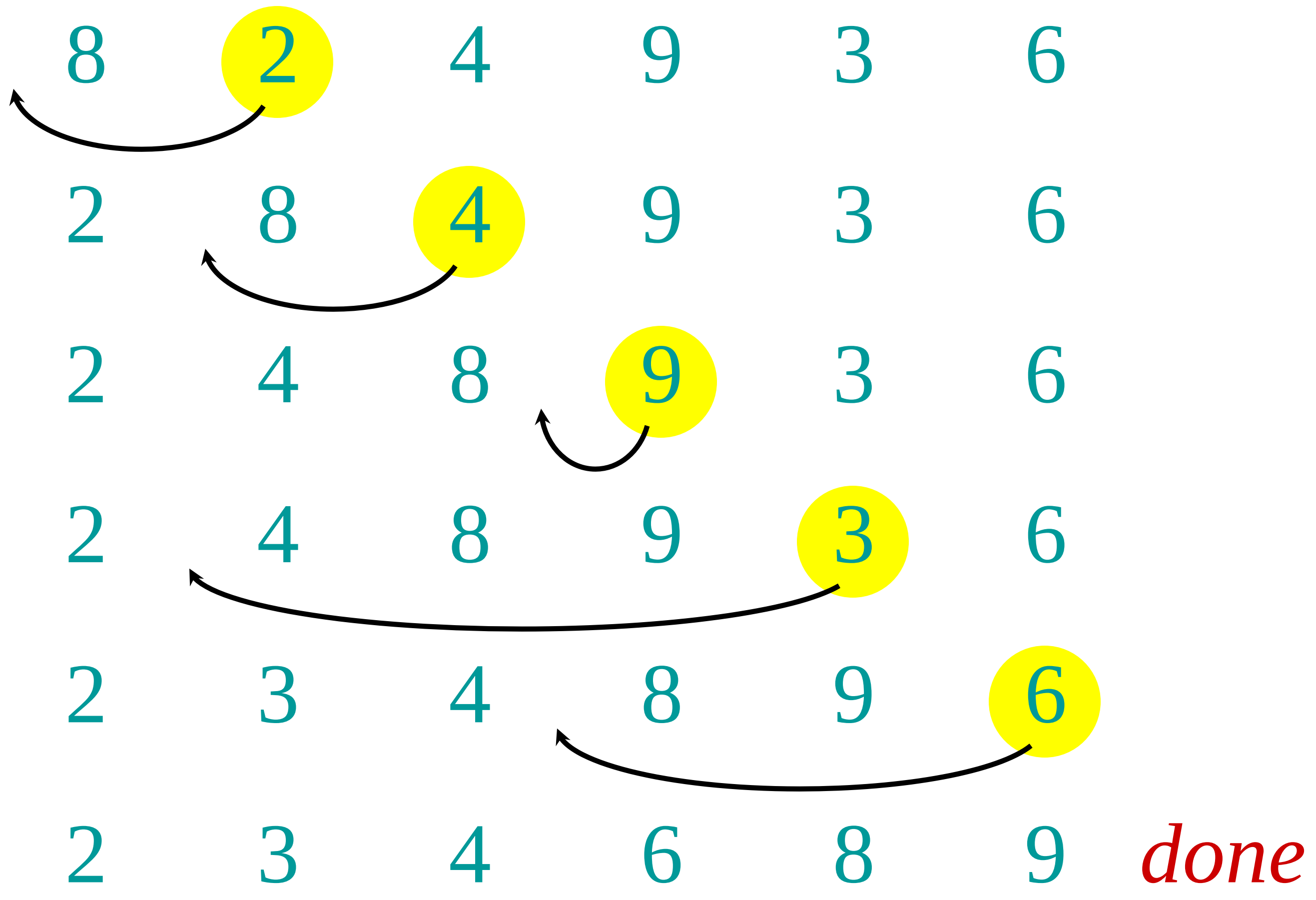
Example of insertion sort



Example of insertion sort



Example of insertion sort



Running time

- The running time depends on the input: an already sorted sequence is easier to sort.
- Major Simplifying Convention: Parameterize the running time by the size of the input, since short sequences are easier to sort than long ones.
  - $T_A(n)$ = time of A on length n inputs
- Generally, we seek upper bounds on the running time, to have a guarantee of performance.

Kinds of analyses

Worst-case: (usually)

- $T(n)$ = maximum time of algorithm on any input of size n.

Average-case: (sometimes)

- $T(n)$ = expected time of algorithm over all inputs of size n.
- Need assumption of statistical distribution of inputs.

Best-case: (NEVER)

- Cheat with a slow algorithm that works fast on *some* input.

Machine-independent time

What is insertion sort's worst-case time?

BIG IDEAS:

- ***Ignore machine dependent constants*,**
  otherwise impossible to verify and to compare algorithms

- Look at ***growth*** of $T(n)$ as $n \to \infty$.

"Asymptotic Analysis"

Θ-notation

DEF:

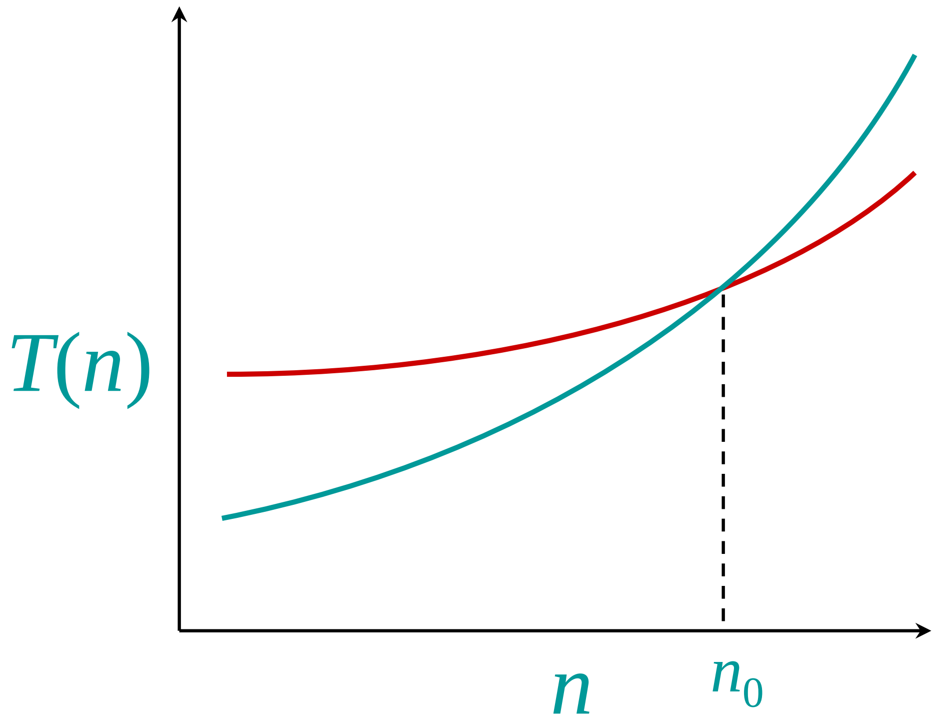
$\Theta(g(n)) = \{ f(n)$: there exist positive constants c_1, c_2, and n_0 such that $0 \le c_1 g(n) \le f(n) \le c_2 g(n)$ for all $n \ge n_0$ $\}$

Basic manipulations:

- Drop low-order terms; ignore leading constants.
- Example: $3n^3 + 90n^2 - 5n + 6046 = \Theta(n^3)$

Asymptotic performance

When n gets large enough, a $\Theta(n^2)$ algorithm *always* beats a $\Theta(n^3)$ algorithm.



- Asymptotic analysis is a useful tool to help to structure our thinking toward better algorithm
- We shouldn't ignore asymptotically slower algorithms, however.
- Real-world design situations often call for a careful balancing

Insertion sort analysis

Worst case: Input reverse sorted.

$$T(n) = \sum_{j=2}^{n} \Theta(j) = \Theta\left(n^2\right) \qquad \text{[arithmetic series]}$$

Average case: All permutations equally likely.

$$T(n) = \sum_{j=2}^{n} \Theta(j/2) = \Theta\left(n^2\right)$$

Is insertion sort a fast sorting algorithm?

- Moderately so, for small n.
- Not at all, for large n.

Example 2: Integer Multiplication

- Let X = $\boxed{A}\boxed{B}$ and Y = $\boxed{C}\boxed{D}$ where A,B,C and D are n/2 bit integers
- Simple Method: $XY = (2^{n/2}A+B)(2^{n/2}C+D)$
- Running Time Recurrence

$$T(n) < 4T(n/2) + 100n$$

- Solution $T(n) = \theta(n^2)$

Better Integer Multiplication

- Let $X = \boxed{A \mid B}$ and $Y = \boxed{C \mid D}$ where A,B,C and D are n/2 bit integers
- Karatsuba:

  $XY = (2^{n/2}+2^{n})AC+2^{n/}2(A-B)(C-D) + (2^{n/2}+1)\ BD$
- Running Time Recurrence

  $T(n) < 3T(n/2) + 100n$

- Solution: $\theta(n) = O(n^{\log 3})$

Example 3:Merge sort

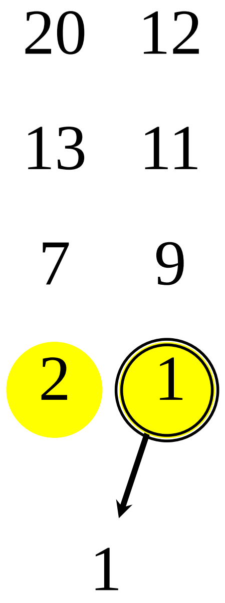
```
MERGE-SORT $A[1 \mathinner{..} n]$
    1. If $n = 1$, done.
    2. Recursively sort $A[1 \mathinner{..} \lceil n/2 \rceil]$
       and $A[\lceil n/2 \rceil + 1 \mathinner{..} n]$.
    3. "Merge" the 2 sorted lists.
```

Key subroutine: **MERGE**

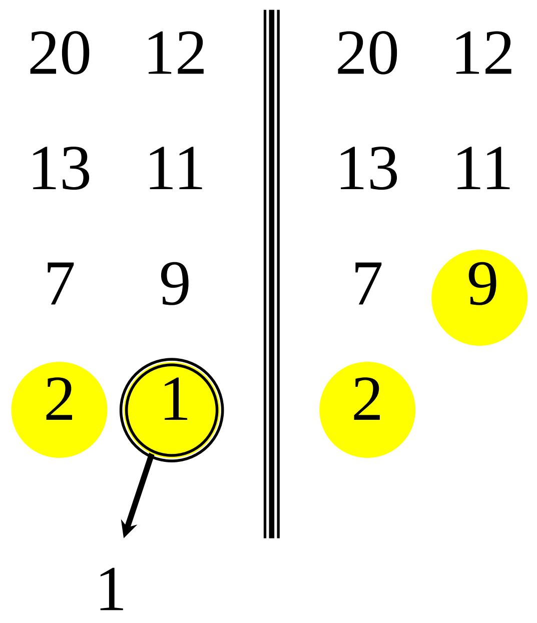
Merging two sorted arrays

20	12
13	11
7	9
2	1

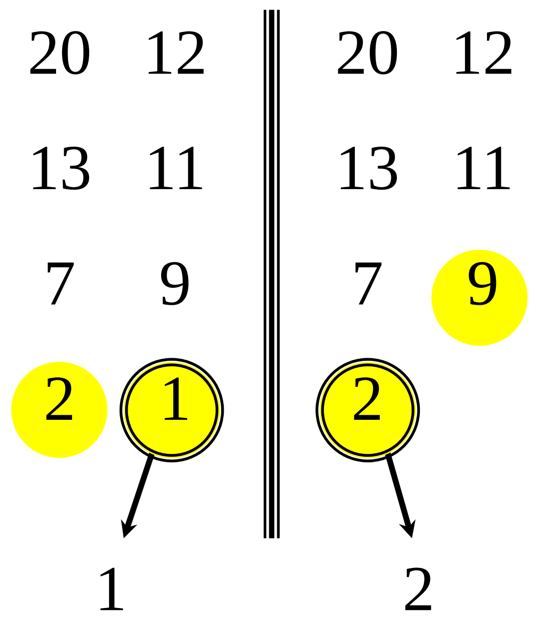
Merging two sorted arrays



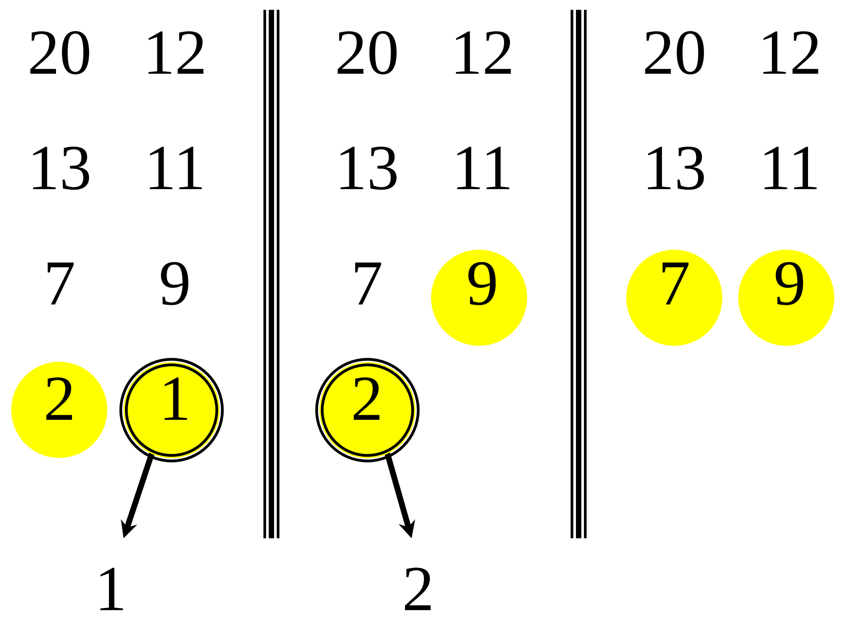
Merging two sorted arrays



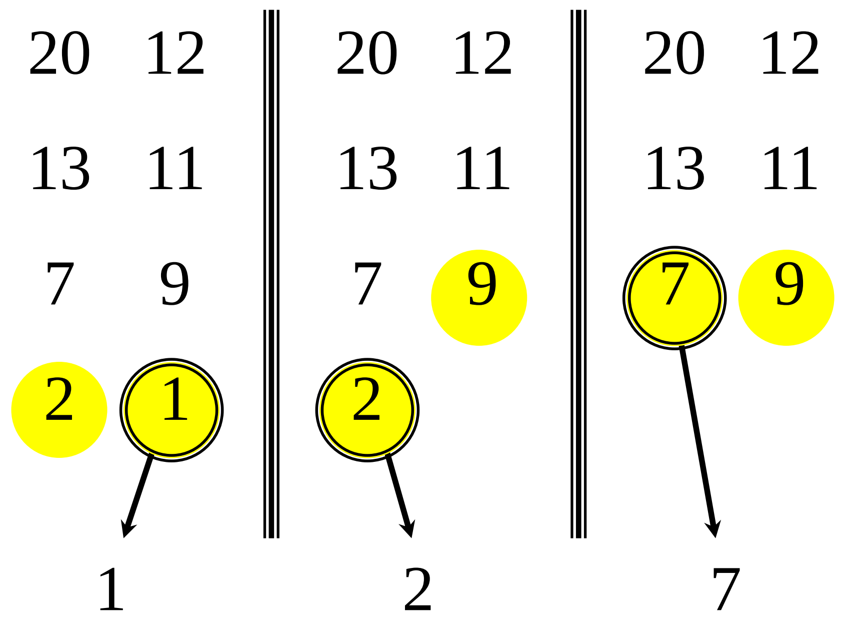
Merging two sorted arrays



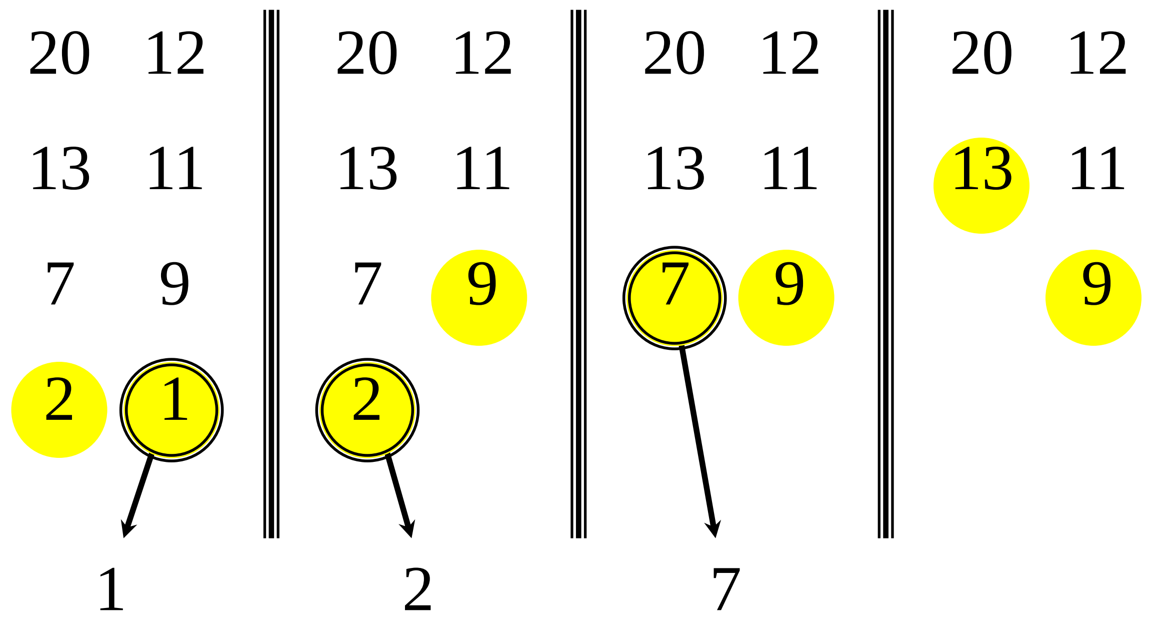
Merging two sorted arrays



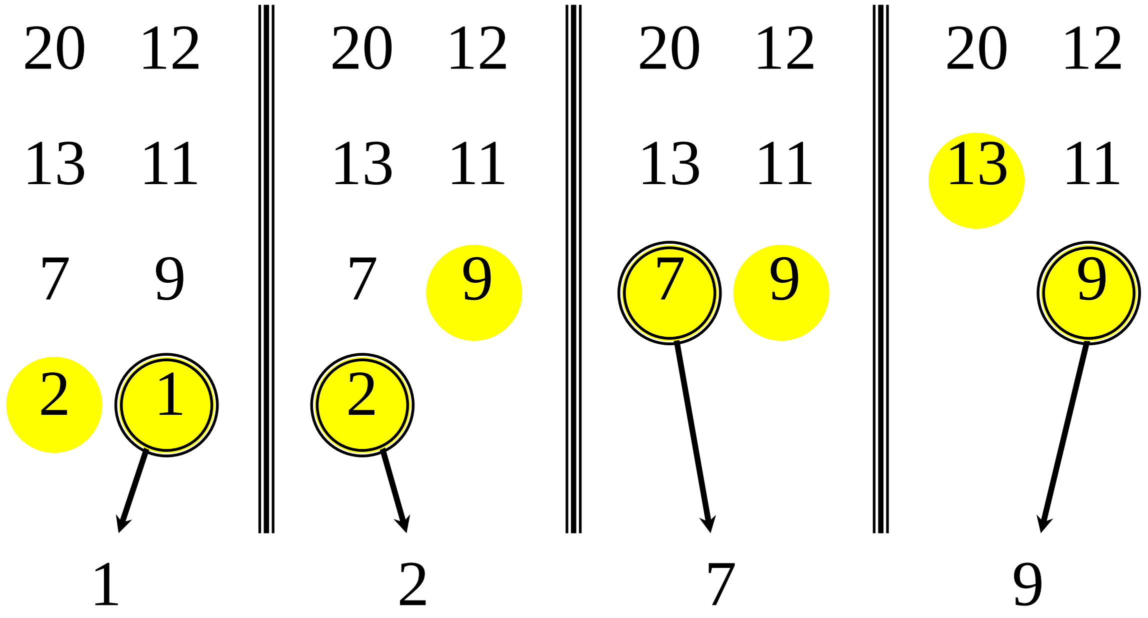
Merging two sorted arrays



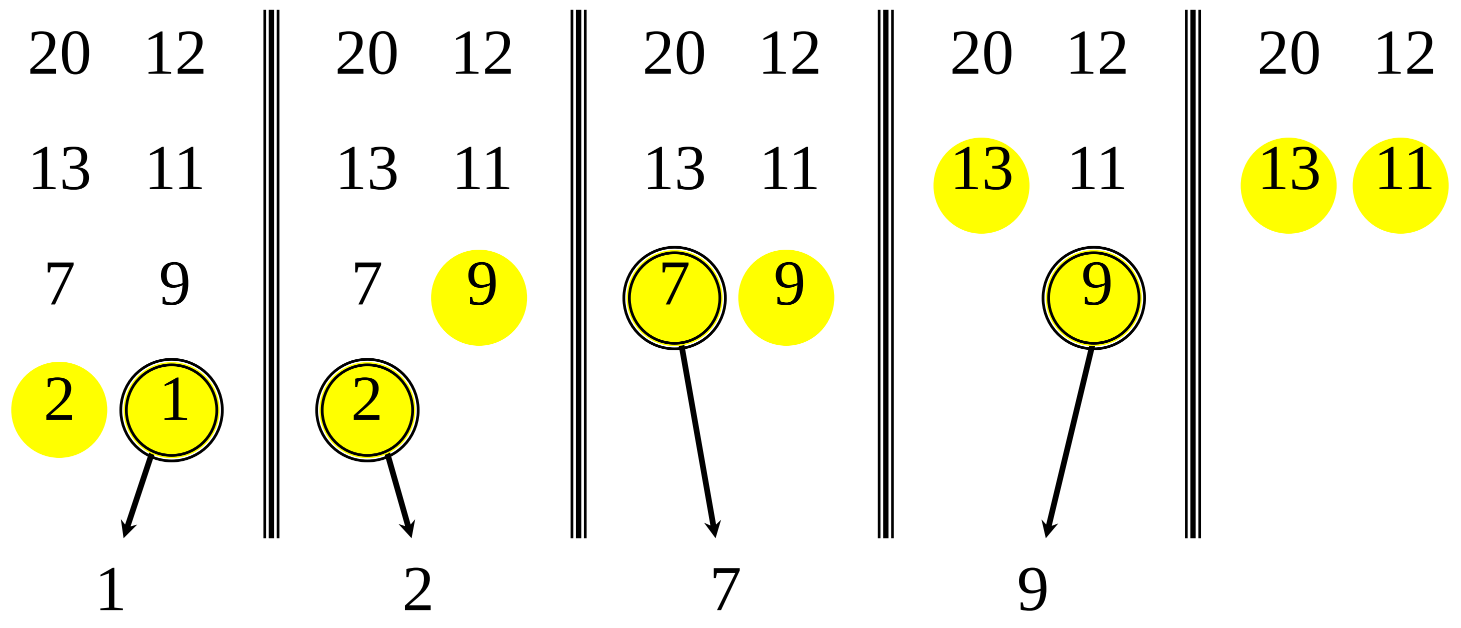
Merging two sorted arrays



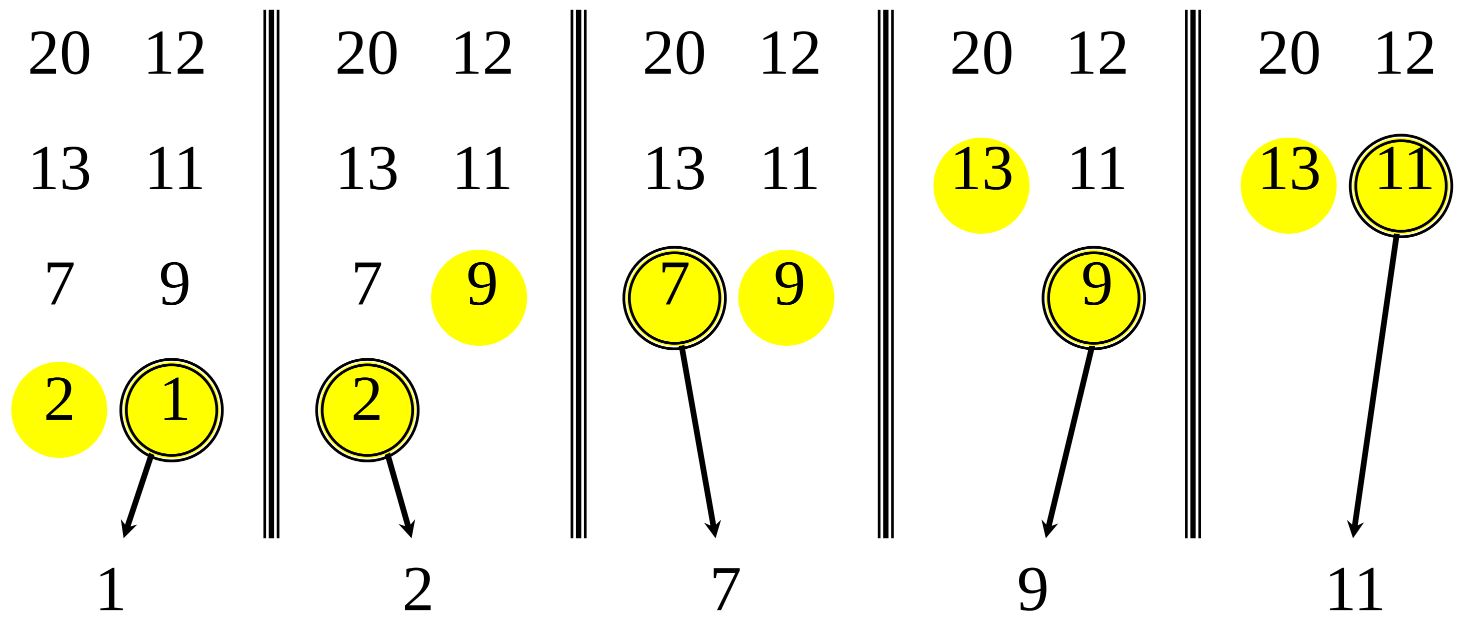
Merging two sorted arrays



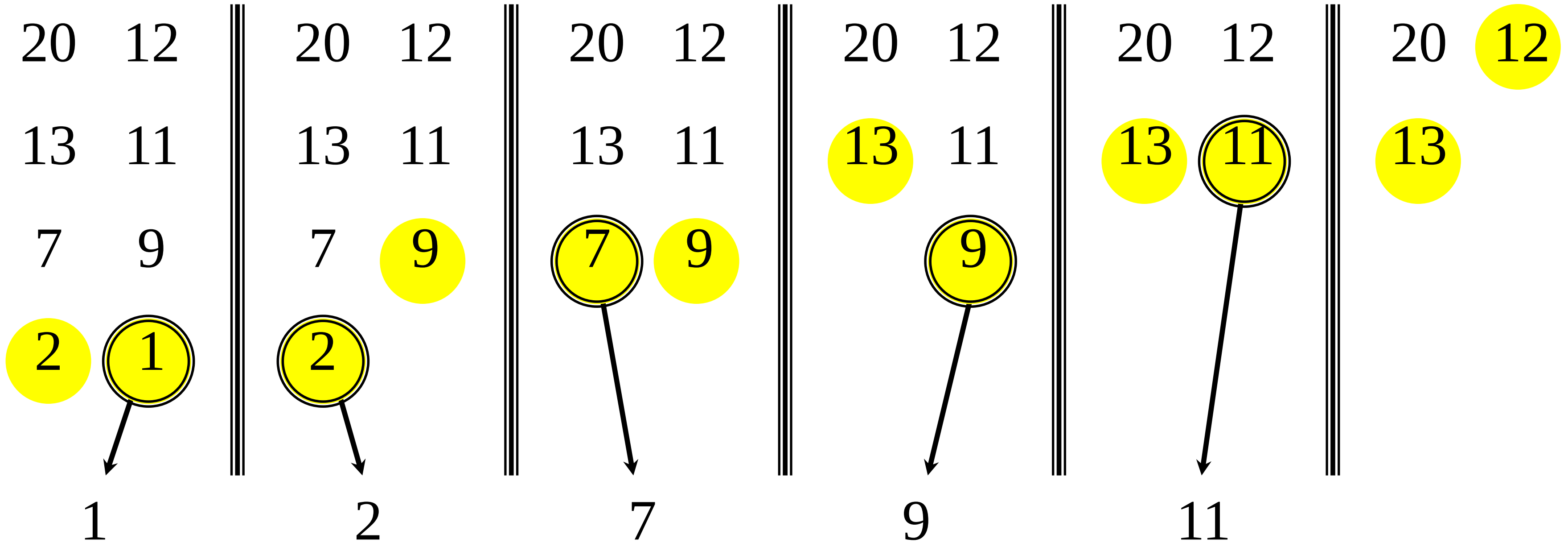
Merging two sorted arrays



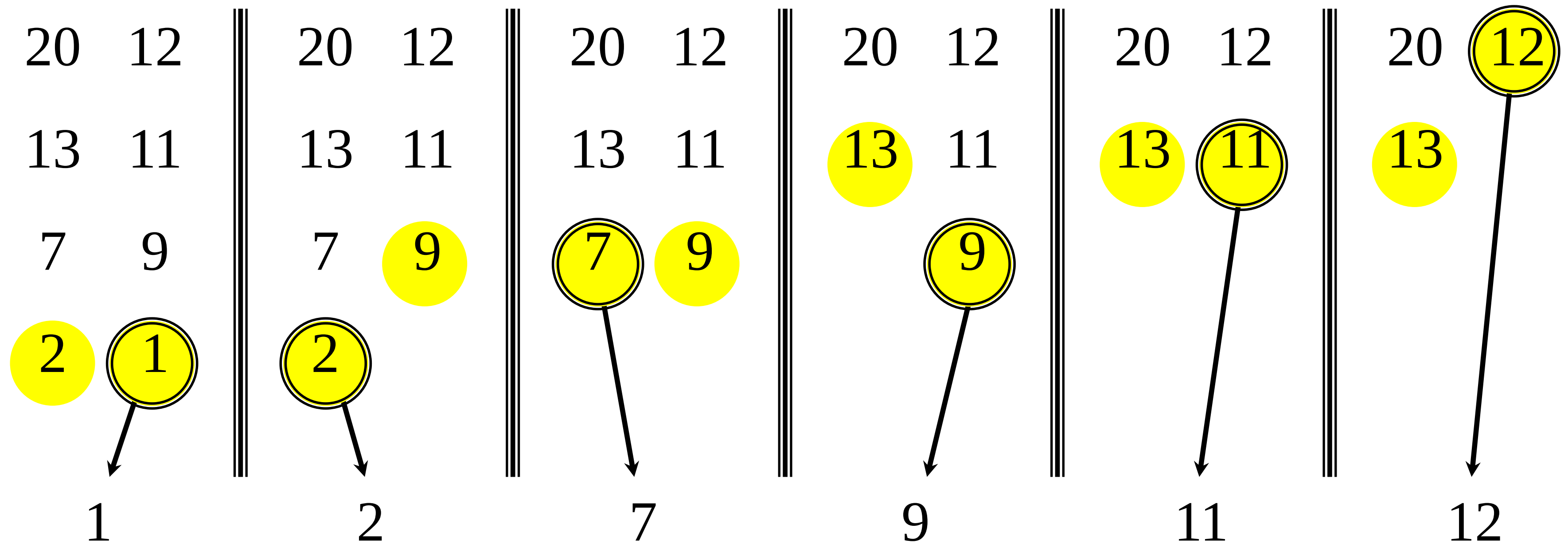
Merging two sorted arrays



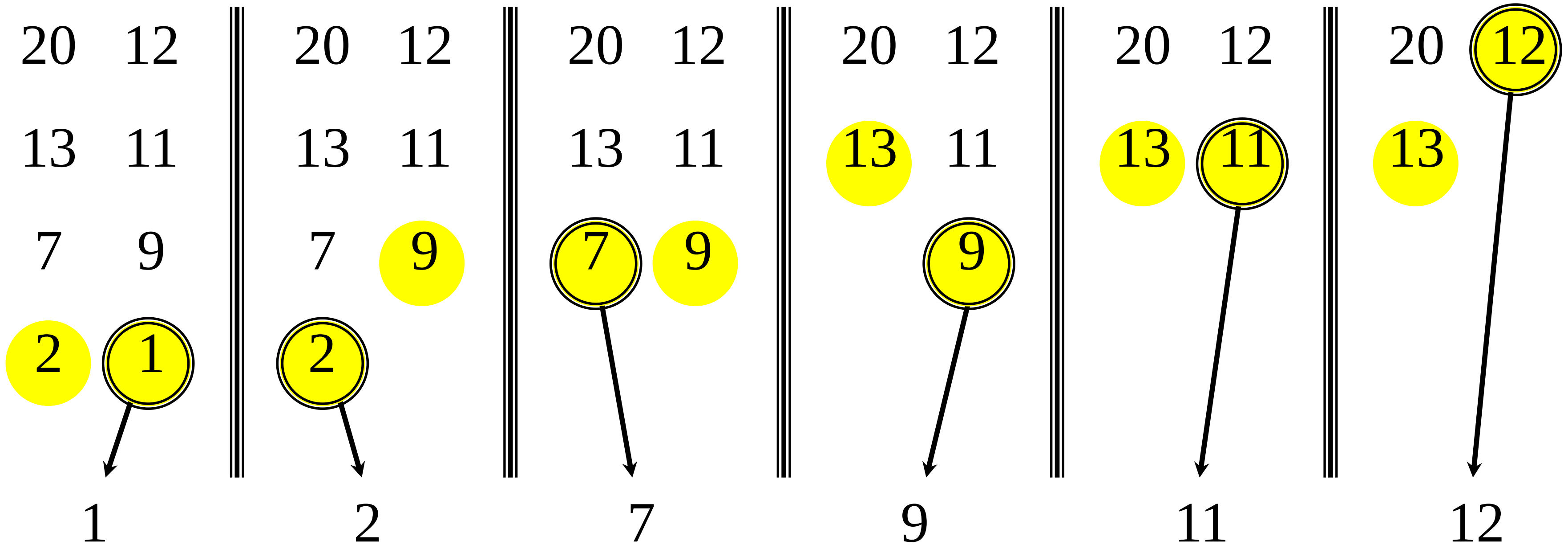
Merging two sorted arrays



Merging two sorted arrays



Merging two sorted arrays



Time $= \Theta(n)$ to merge a total of n elements (linear time).

Analyzing merge sort

$T(n)$	MERGE-SORT $A[1\,..\,n]$
$\Theta(1)$	1. If $n = 1$, done.
$2T(n/2)$	2. Recursively sort $A[\,1\,..\,\lceil n/2\rceil\,]$ and $A[\,\lceil n/2\rceil+1\,..\,n\,]$.
$\Theta(n)$	3. ***"Merge"*** the 2 sorted lists

Sloppiness: Should be $T(\lceil n/2\rceil) + T(\lfloor n/2\rfloor)$, but it turns out not to matter asymptotically.

Recurrence for merge sort

$$T(n) = \begin{cases} \Theta(1) \text{ if } n = 1; \\ 2T(n/2) + \Theta(n) \text{ if } n > 1. \end{cases}$$

- We shall usually omit stating the base case when $T(n) = \Theta(1)$ for sufficiently small n, but only when it has no effect on the asymptotic solution to the recurrence.
- Lecture 2 provides several ways to find a good upper bound on $T(n)$.

Recursion tree

Solve $T(n) = 2T(n/2) + cn$, where $c > 0$ is constant.

Recursion tree

Solve $T(n) = 2T(n/2) + cn$, where $c > 0$ is constant.

$$T(n)$$

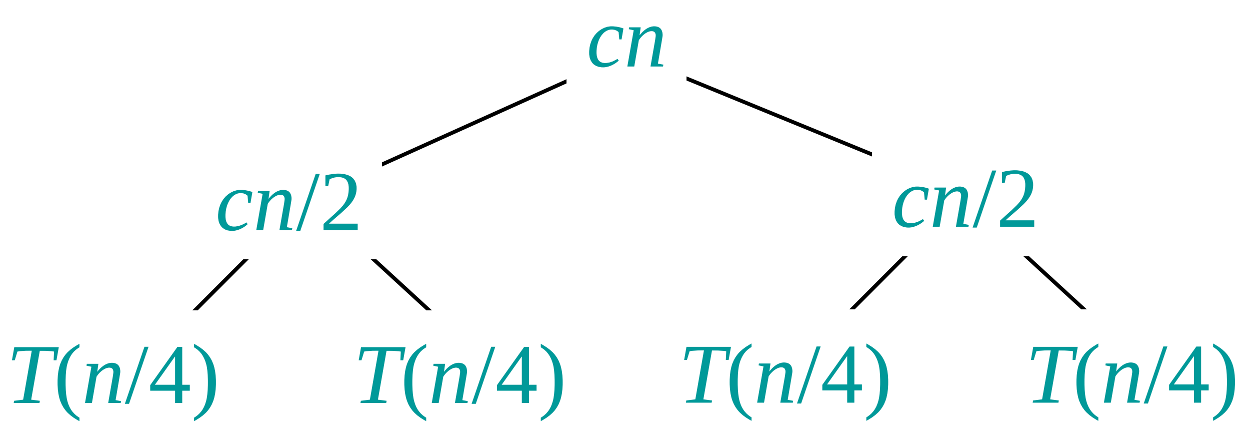
Recursion tree

Solve $T(n) = 2T(n/2) + cn$, where $c > 0$ is constant.

$$\begin{tikzcd} & cn \arrow[dl, no head] \arrow[dr, no head] & \\ T(n/2) & & T(n/2) \end{tikzcd}$$

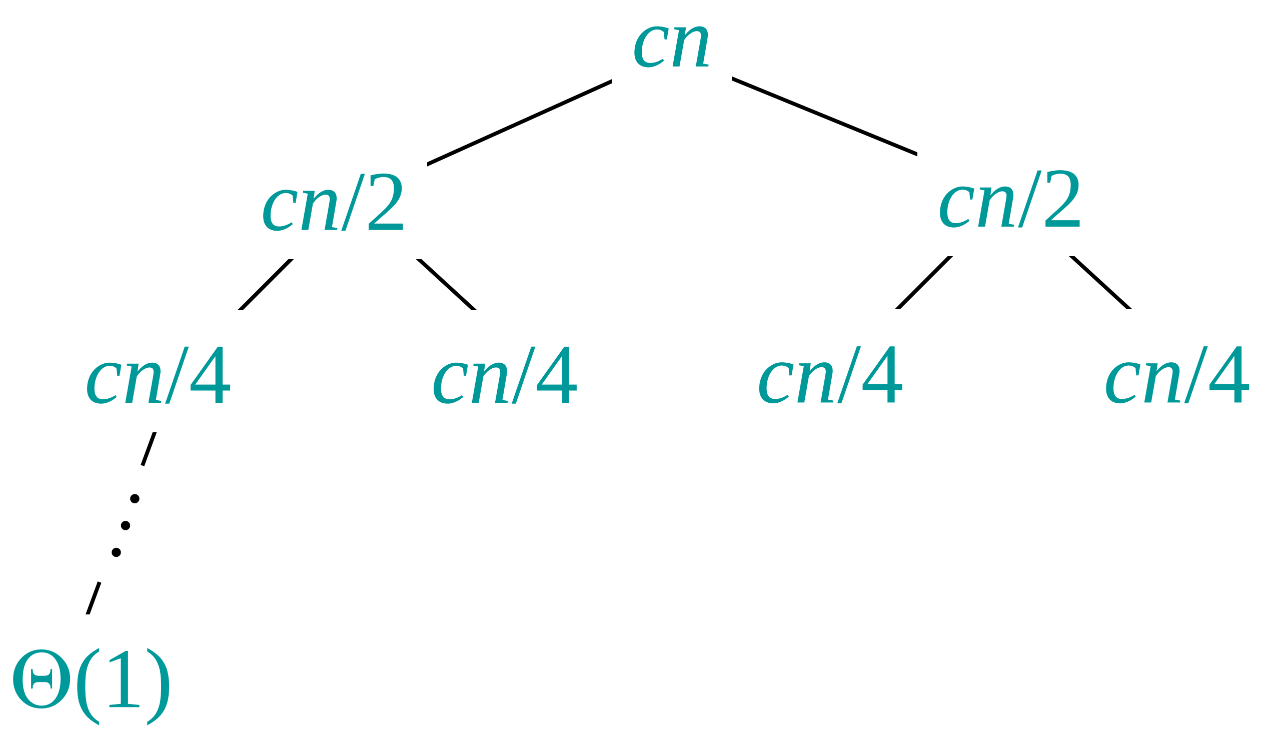
Recursion tree

Solve $T(n) = 2T(n/2) + cn$, where $c > 0$ is constant.



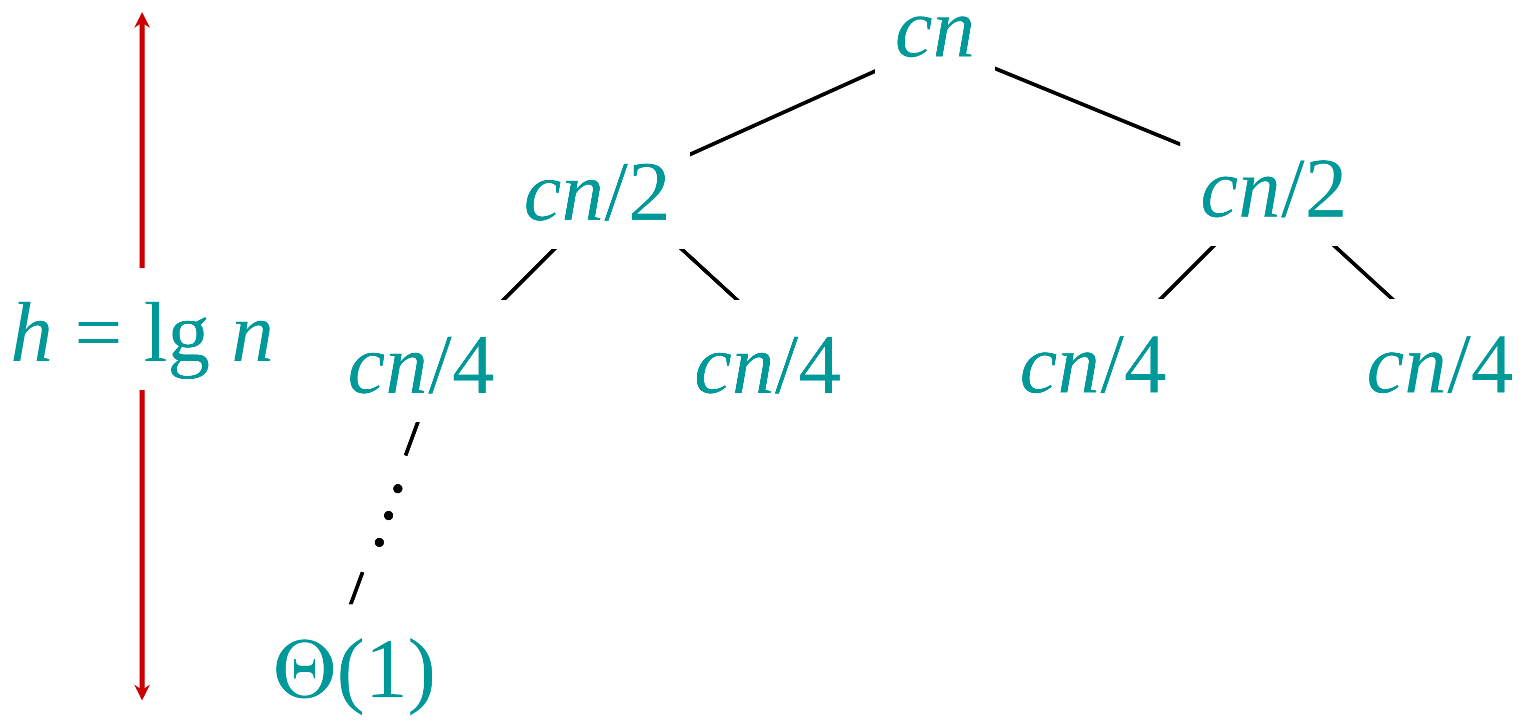
Recursion tree

Solve $T(n) = 2T(n/2) + cn$, where $c > 0$ is constant.



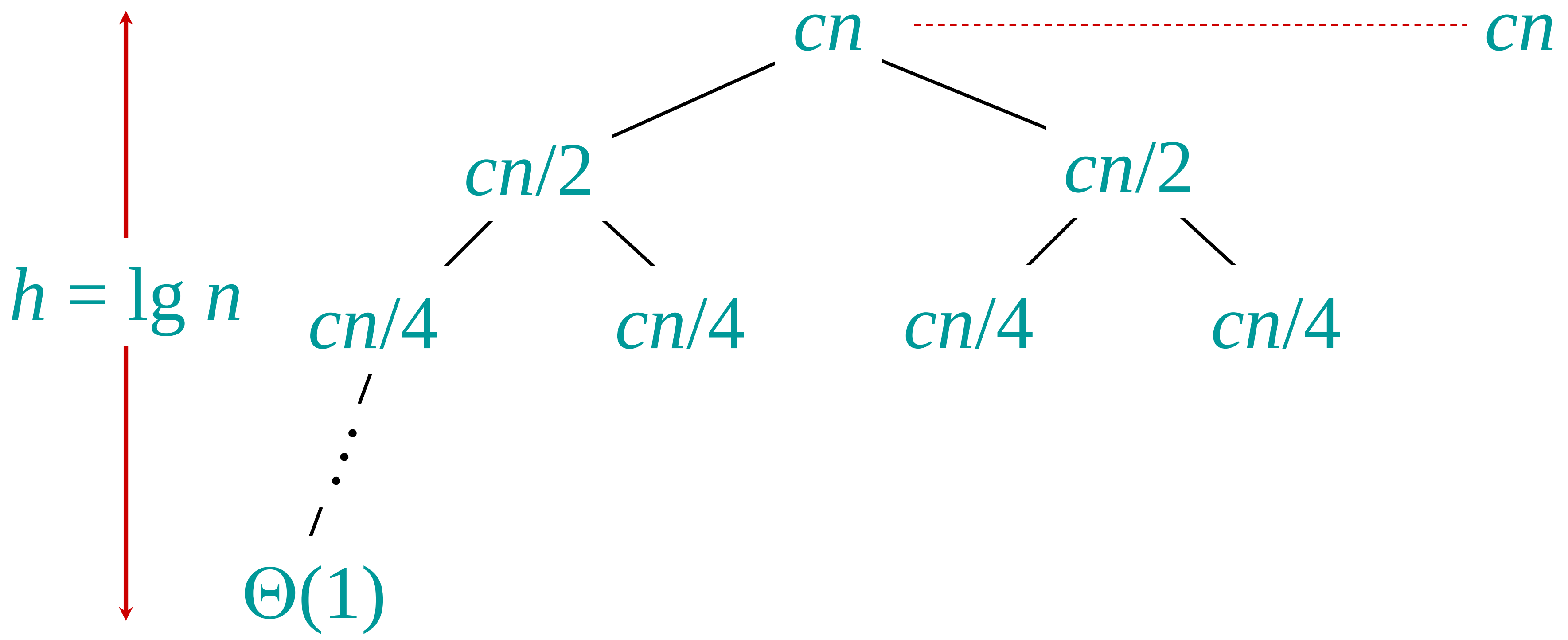
Recursion tree

Solve $T(n) = 2T(n/2) + cn$, where $c > 0$ is constant.



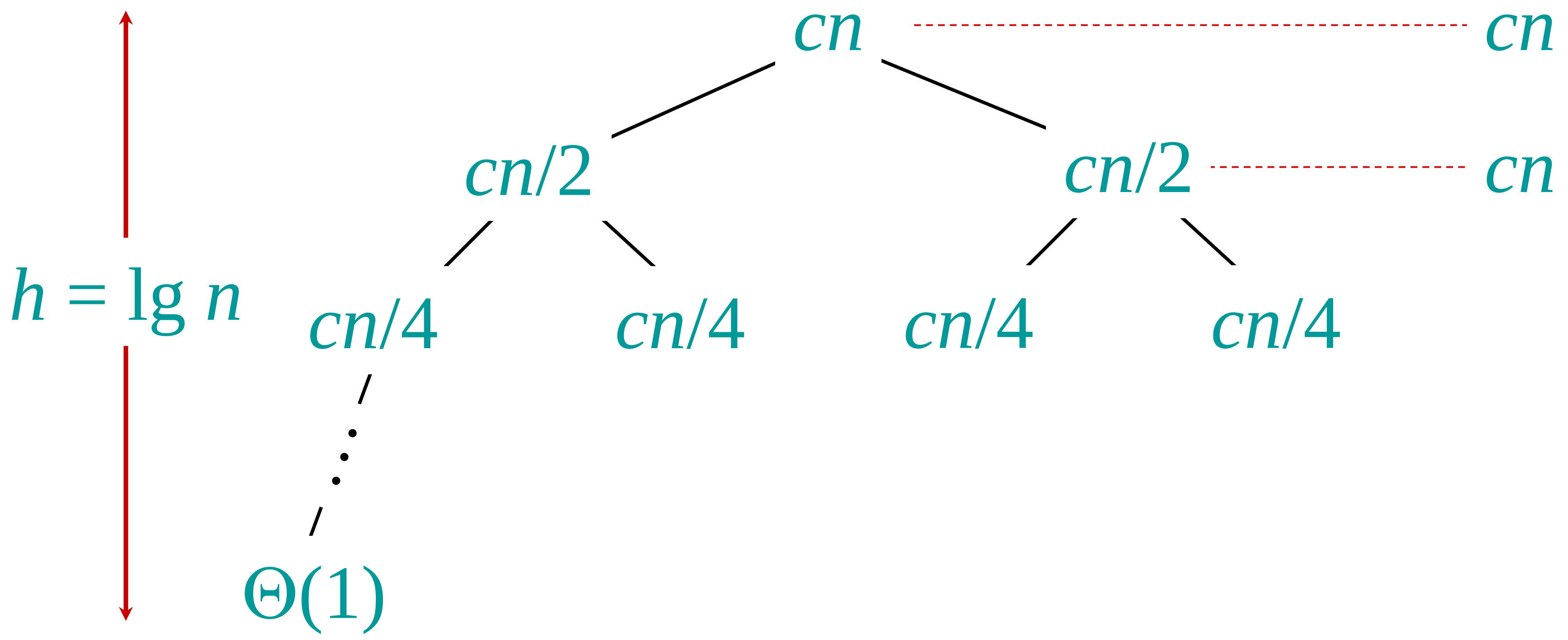
Recursion tree

Solve $T(n) = 2T(n/2) + cn$, where $c > 0$ is constant.



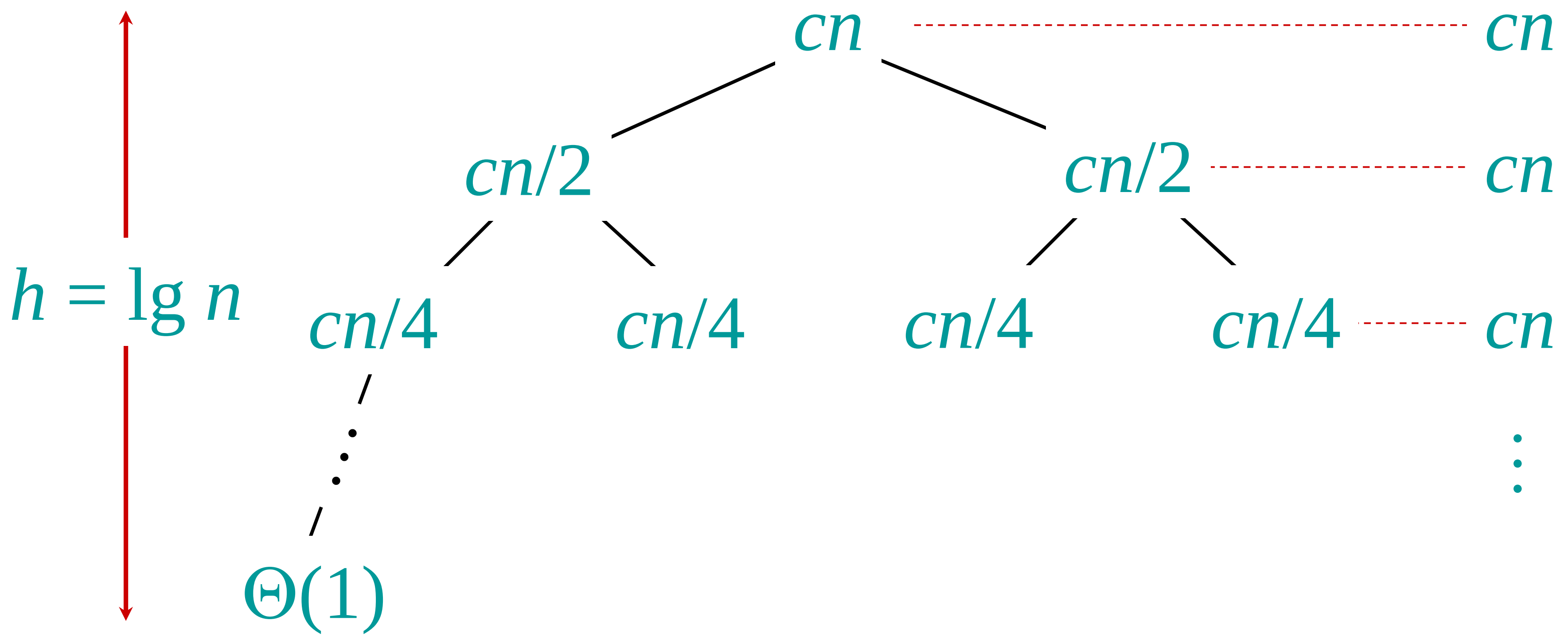
Recursion tree

Solve $T(n) = 2T(n/2) + cn$, where $c > 0$ is constant.



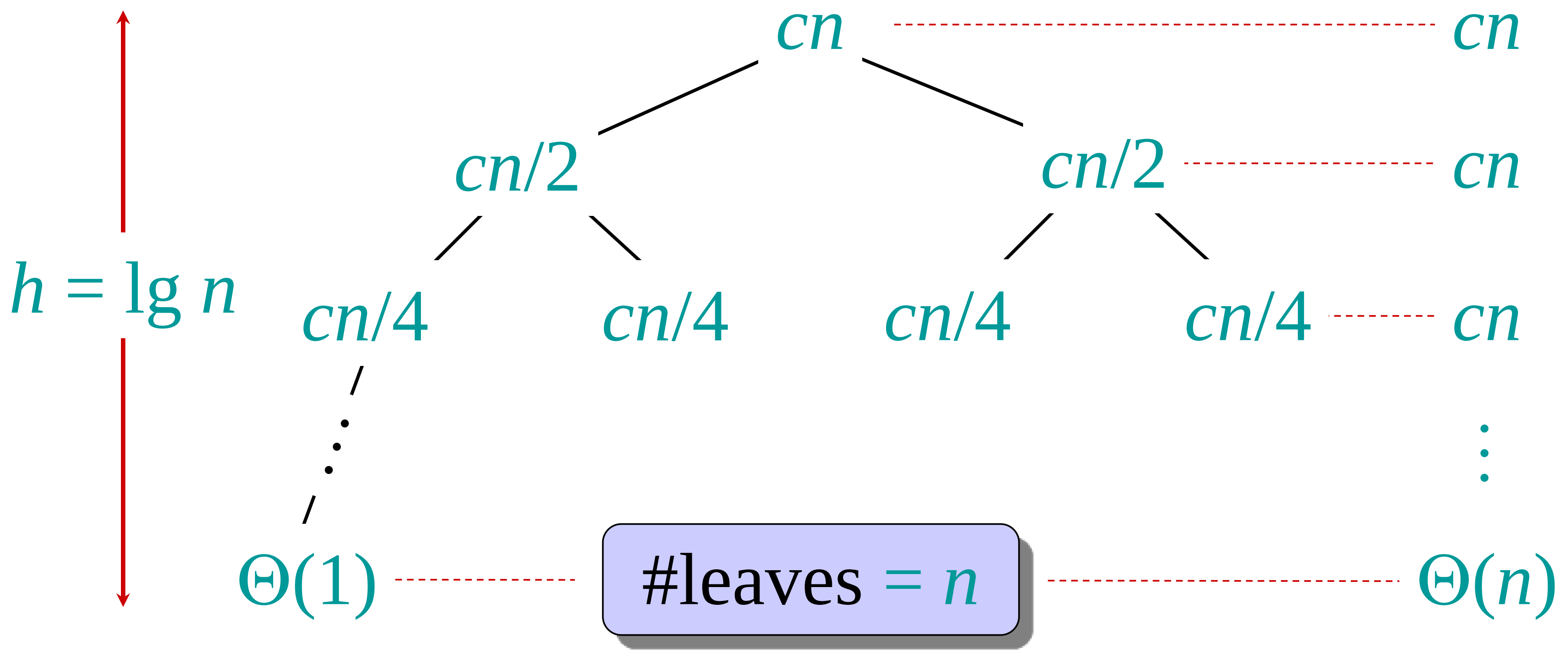
Recursion tree

Solve $T(n) = 2T(n/2) + cn$, where $c > 0$ is constant.



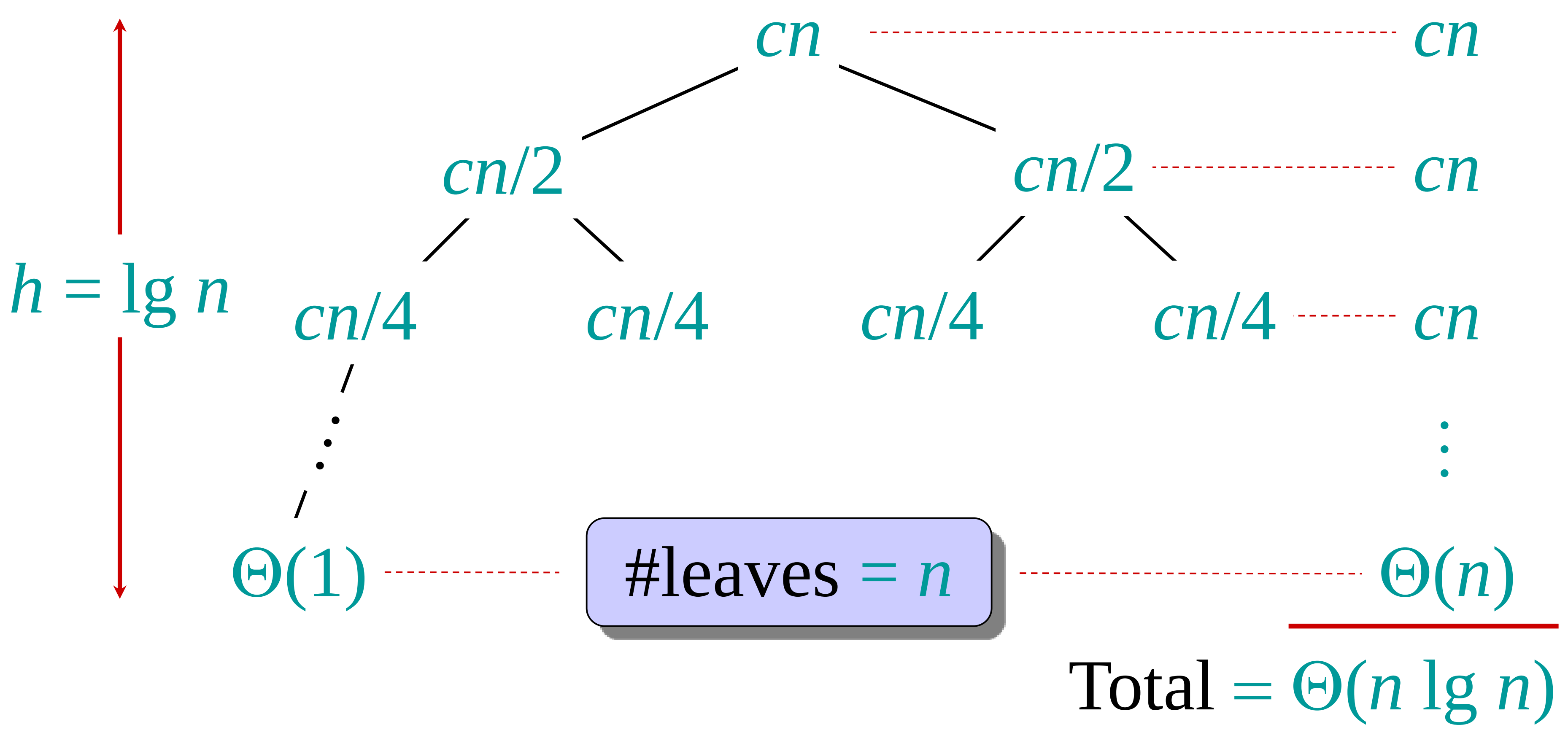
Recursion tree

Solve $T(n) = 2T(n/2) + cn$, where $c > 0$ is constant.



Recursion tree

Solve $T(n) = 2T(n/2) + cn$, where $c > 0$ is constant.



Conclusions

- $\Theta(n \lg n)$ grows more slowly than $\Theta(n^2)$.
- Therefore, merge sort asymptotically beats insertion sort in the worst case.
- In practice, merge sort beats insertion sort for $n > 30$ or so.

ALGORITHM ANALYSIS AND DESIGN

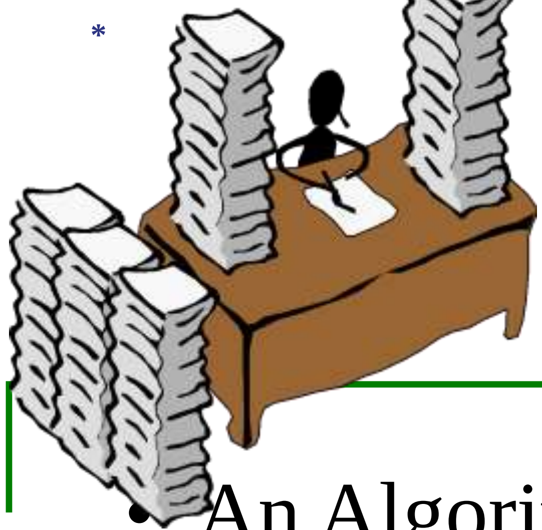
INTRODUCTION

Algorithm Analysis – Asymptotic Notations - Divide and Conquer – Merge Sort – Binary Search - Greedy Algorithms – Knapsack Problem – Dynamic Programming – Warshall's Algorithm for Finding Transitive Closure – Backtracking – Sum of Subset Problem – Branch and Bound – Travelling Salesman Problem.

Algorithm

- An algorithm is a step by step procedure for solving the given problem.

- An algorithm is independent of any programming language and machine.



Definition

- An Algorithm is a finite sequence of effective steps to solve a particular problem where each step is unambiguous(definite) and which terminates for all possible inputs in a finite amount of time.
- An Algorithm accepts zero or more inputs and produces one or more outputs.

Properties

- Definiteness
- Effectiveness
- Finiteness

Definiteness

- Each instruction should be clear and unambiguous.
- It must be perfectly clear what should be done.
- Example Directions which are not permitted
  - "add 6 or 7 to x"
  - compute x/0
  - remainder of m/n when m and n are –ve numbers
- it is not clear which of the 2 possibilities should be done.

Definiteness Contd..

- Achievement using programming language for algorithms
  - designed such that each legitimate sentence has a unique meaning

Effectiveness

- Each step must be such that it can at least in principle be done by a person using paper and pencil in a finite amount of time.
- Example -Effective
  - Performing arithmetic on integers
- Example not effective
  - Some arithmetic with real numbers.
  - *if k is the largest integer n such that x^n + y^n = z^n in all positive integers then go to step 4*

Finiteness

- The algorithm should terminate after a finite number of steps in all the cases.
- The time for termination should be reasonably short.

Example Algorithm

- An algorithm to find the maximum in array of n numbers

```
Algorithm max(a,n)
// a is an array of size n
{  result:= a[1];
  for i = 2 to n do
 { if (a[i] > result)
 then result :=a[i];
}
  return result;
}
```

Development of an algorithm

- Steps Involved
  - Devise
  - Validate
  - Analyse
  - Test

Devise the algorithm

- An art which cannot be fully automated
- Design techniques
  - Yields good algorithms for known problems
  - Helps to devise new useful algorithms

General Techniques

- Provide directions for algorithmic problem solutions
- Help programmers thinking in designing an algorithm
- Types - depending on applied problems
  - Solve almost all variety problems.
  - Solve only to specific type of problems.

Devise Algorithm For New Problems

- Requires exploration of the specific features of the problem discussed
- General design techniques can be applied to a much extend.
- The best technique has to be selected from the possible set of techniques.
- It should be applied in the right way.

Example Sorting Approaches

- Incremental
  - Insertion sort
- Divide-and-conquer
  - Quick sort
  - Merge sort etc.

Validating the algorithm

- To show that algorithm computes the correct answer for all possible legal inputs.
- Need not be expressed as a program.
  - sufficient to state it any precise way.
- Assurance of correctness of algorithm
  - Independent of the issues concerning the programming language

Validating the algorithm

- A proof of correctness is more valuable than 100 tests.
- Mathematical proof
  - solution stated in two forms
- Program
  - The program is expressed as a set of predicate calculus assertions in input and output variables.
- Specification
  - specifications are also expressed as predicate calculus assertions

Analyse algorithms

- This phase performs performance analysis
- Execution requires CPU time and memory
- Task of determining time and space requirements.
- Performance in cases considered separately
  - Best case
  - Worst case
  - Average case etc.

Good algorithm

- One which works correctly
  - should give the desired output
- Readable
  - Steps are clear and understandable
- Efficient in terms of time and memory utilization .
  - give the desired output faster
  - utilize less resources.
- Given multiple solutions, finding out the better one.

Performance Measures

- Quality of the solution
  - How much time does the solution take?.
  - How much space does the solution occupy?
- Simplicity of the solution
- Performance Improvement
  - Improving the algorithm design.
  - Continuous improvements in hardware and communication infrastructure

Testing

- Two phases
  - Debugging
  - Profiling (performance measurement)

Debugging

- Process of executing programs on sample data sets to determine whether faulty results occur and if so to correct them.
- Requirement
  - A proof of correctness prove that the algorithm will correctly for all cases.
  - Not easy to obtain mathematical proofs.

Profiling

- Executing programs with sample data sets to measure the time and space requirements.
- It is helpful for optimization since it can point out logical places that require optimization.

Basic Mathematical Principles
A few formulae

$1 + 2 + \dots + n = n * (n + 1) / 2$

$1 + 2^2 + \dots + n^2 = n * (n + 1) * (2n + 1) / 6$

$1 + a + a^2 + \dots + a^n = (a^{(n+1)} – 1) / (a – 1)$

$a+ar+ar^2+ar^3+\dots\dots\dots+ar^n=a(r^{n+1}-1)/(r-1)$

$$\sum_{I=1}^{n} I^k \approx \frac{1}{k+1} \times n^{k+1}$$

$$\sum_{I=1}^{k} i \times 2^i = (k-1) \times 2^{k+1} + 2$$

Mathematical Formulae Contd..

- Floor(x) or $\lfloor x \rfloor$ is the largest integer less than or equal to x
- Ceil(x) or $\lceil x \rceil$ is the smallest integer greater than or equal to x

Mathematical Formulae Contd..

$\log_a xy = \log_a x + \log_a y$

$\log_a (x/y) = \log_a x - \log_a y$

$\log_a x^n = n \log_a x$

$\log_a 1 = 0$

$\log_x x = 1$

$\log_a x = 1/\log_x a$

$\log_a(x) = \log_a(b) * \log_b(x)$

$\log_a x = \log_b x / \log_b a$

$a^{\log_a x} = x$

if $2^k = n$ then $k = \log_2 n$

$x^{\log_b y} = y^{\log_b x}$

Mathematical Formulae Contd..

$$C(n,k) = \binom{n}{k} = \frac{n \times (n-1) \times - - \times (n-k+1)}{k!}$$

$$C(n,k) = \binom{n}{k} = \frac{n!}{(n-k)! \times k!}$$ **for n>=k>=0**

$$\sum_{k=0}^{n} C(n,k) = 2^n$$

Mathematical Formulae Contd..

- A.P.
  - Nth term - $a + (n - 1)d$
  - Sum - $Sn = n/2*(2a + (n - 1)d)$
- *G.P.*
  - Nth term - $ar^{(n-1)}$
  - Sum - $Sn = a(1 - r^n)/(1 - r)$ (valid only if $r \neq 1$)

A Commonly Used Result During Analysis

- How many times should we half the number n (discarding reminders if any) to reach 1?
- two cases.
- **Case – 1**: n is a power of 2
- **Case – 2**: n is not a power of 2

A Commonly Used Result Contd..

Case – 1: n is a power of 2 → $n = 2^m$

Example n = 8

8 must be halved 3 times to reach 1

8 4   2   1

16 must be halved 4 times to reach 1

16   8   4   2   1

A Commonly Used Result Contd..

- **Case – 2**: n is not a power of 2 ; n > 2^m

Example n = 9

9 must be halved 3 times to reach 1

9 4 2   1

15 must be halved 3 times to reach 1

15   7   3   1

A Commonly Used Result Contd..

General Case $2^m \le n < 2^{(m+1)}$

n must be halved m times to reach 1

n must be halved m times if $2^m \le n < 2^{(m+1)}$

So $m \le \log_2(n) < m+1$

$m = \text{floor}(\log_2 n)$ or $\lfloor \log_2 n \rfloor$

Number n must be halved $\lfloor \log_2 n \rfloor$ times to reach 1

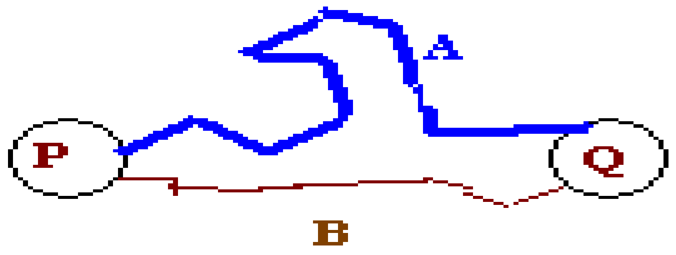
A Corollary to the result Contd..

- A number n must be halved floor($\log_2 n$) + 1 times
- To reach 0

Performance Analysis

- to improve existing algorithms
- to choose among several available algorithms
- Two types
  - Apriori Analysis
  - Aposteriori Analysis

Apriori Analysis - Example



- Doing an analysis of the solutions before performing the action.
- Physical Example- Find path from P to Q
- Criteria for selection
  - Path length, road conditions, type of vehicle, speed

Apriori Analysis

- Doing an analysis of the solutions before coding the algorithms
- Given two or more algorithms for a problem
  - Doing a machine independent analysis to find better algorithm

Space Time Tradeoff Example

Store Employee information

- Solutions
  - Array
  - Linked list
- Which locates a particular employee faster?
- Which provides better utilization of memory?
- Which is easy to code?
- Which is easier to test?

Space Complexity

- Amount of memory the algorithm needs to run to completion
- Requirements
  - Fixed part
  - Variable part
- S(P)=c+Sp

Example Algorithm

```
Algorithm abc(a,b,c)
{return a+b+b*c+(a-c)/(a+b) +2.0;
}
```

S(P) = c + Sp

Sp = 0 S(P)>3

Example Algorithm

```
Algorithm sum(a,n)
{s = 0.0;
for i:=1 to n do s:=s+a[i];
return s;  }
```

one word for each integer n,i,s,a[]

s(n) >= n+3

Example Algorithm

```
algorithm rsum(a,n)
{ if (n<=0) then return 0.0
else return rsum(a,n-1)+a[n]; }
```

- Instance characterized by n
- Recursive stack space required at each level
    - (n,return address,*a[]) 3 words
- Depth of recursion considering first invocation - (n+1)
- S(n)>=3*(n+1)

Time Complexity

- Amount of computer time needed for the algorithm to run to completion
- T(P) = C(P) + R(P)
- C(P) is independent of problem instance and is always not required (recompilation)
- So usually only R(P) is considered

Apriori Measures

- Instruction count
  - Basic operations
  - Basic steps
- Size of input.

Computing Instruction Count

- Initialization instructions
- Loops.
  - Number of passes of the loops
- Count the basic operations/steps

Basic Steps - Example

- Search
  - Basic operation is compare x and y
- Matrix multiplication
  - Basic operation is multiplication of real numbers

Step in Algorithm

- Step is a meaningful segment of program or computational unit that is independent of the instance characteristics
- Example
  - 10 or 100 additions can be one step
  - 200 multiplications can be another step
  - but not n additions

Example 1

```
Algorithm sum(a,n)
{s=0.0;
count:=count+1;
for i: =1 to n do
    {count:=count+1;
    s:=s+a[I];
    count:=count+1;}
count:=count+1;
count:count+1;
    return s;}
```

count = 2n+3

Example 2

```
algorithm rsum(a,n)
{
count:=count+1;
if (n<=0) then
{ count:=count+1;
return 0.0; }
else
{ count:=count+1;
   return rsum(a,n-1)+a[n];} }
```

Count ?

Computing Step Count from s/e

- Each statement will have an (s/e) steps/execution depending on the type of statement
- Frequency of each statement is multiplied with the corresponding s/e of that statement to get the step count for that statement.
- Statement Step counts are added to give the total count

$$\sum f \times (s/e)$$

Example 1

```
Algorithm sum(a,n)
{ s=0.0;
for i: =1 to n do
{ s:=s+a[i];   }
return s;       }
```

s/e	freq	total
1	1	1
1	n+1	n+1
1	n	n
1	1	1

		2n+3

Example 2

	S/e	freq	Total
algorithm add(a,b,b,m,n)			
{for i:=1 to m do	1	m+1	m+1
for j:=1 to n do	1	m(n+1)	m(n+1)
c[i,j]:=a[i,j]+b[i,j]; }	1	mn	mn

			2mn+2m+1

Example 3

```
algorithm rsum(a,n)
{if (n<=0) then
{ return 0.0; }
else
{return rsum(a,n-1)+a[n];
   }}
```

s/e	freq n=0	freq n>0	total n=0	total n>0
1	1	1	1	1
1	1	0	1	0
1+x	0	1	0	1+x
			----------	----------
			2	2+x

Computing Complexity

$X = t_{rsum}(n-1)$

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

Known as a recurrence relation.

Solving the Recurrence

$t_{rsum}(n)=2+ t_{rsum}(n-1)$

$= 2+2+ t_{rsum}(n-2)$

$= 2*2+ t_{rsum}(n-2)$

$= 2*3+ t_{rsum}(n-3)$

- - - - - - - - - - -

$=2*n + t_{rsum}(n-n)$

$= 2n+2 \qquad n>=0$

Complexity - Representation

- when the step count cannot be uniquely represented, the step count for the best, worst and average cases can be separately listed.

Worst Case Complexity

- Dn be the set of inputs of size n
- i ε Dn.
- Let T(i) be the number of basic operations performed by the algorithm on input i.
- Worst case complexity W can be defined as
  - $W(n) = \max\{ T(i) / i \in Dn\}$

Best Case Complexity

- Dn be the set of inputs of size n
- i ε Dn.
- Let T(i) be the number of basic operations performed by the algorithm on input i.
- Best case complexity B can be defined as
  - B(N)= min{ T(i) / i ε Dn}

Average Case Complexity

- Dn be the set of inputs of size n
- T(i) be the number of basic operations performed by the algorithm on input i.
- P(i) be the probability for input I to occur
- Average behavior can be defined as
- $A(n) = \sum_{i \varepsilon Dn} P(i) \times T(i)$

Computing Average Case Complexity

- Two steps
  - An understanding of the average nature of the input
  - Performing a run-time analysis of the algorithm for different cases
- Difficult to estimate the statistical behavior of the input

Expressing Complexity

- Goodness of an algorithm often expressed in terms of its worst-case running time.
- Two reasons for this:
  - the need for a bound on one’s pessimism
  - ease of calculation of worst-case

Example Algorithm

```
Algorithm insert(A,i,k)
//To insert k at position i in a[1..n]
{Copy a[i…n-1] to a[i + 1…n]
Copy k to a[i] }
```

Worst Case Complexity

```
Algorithm insert(A,i,k)
//To insert k at position i in a[1..n]
{Copy a[i…n-1] to a[i + 1…n]      n-1 copies
Copy k to a[i] }
                                    1 copy
```

total number of copy operations is n-1+1=n.

worst case complexity of array insertion is n steps

Best Case Complexity

```
Algorithm insert(A,i,k)
//To insert k at position i in a[1..n]
{Copy a[i…n-1] to a[i + 1…n]     0 copies
Copy k to a[i] }
                                  1 copy
```

total number of copy operations is 0+1=1

worst case complexity of array insertion is 1 steps

Average Case Complexity

```
Algorithm insert(A,i,k)
//To insert k at position i in a[1..n]
{Copy a[i…n-1] to a[i + 1…n]
Copy k to a[i] }
```

Probability for 0 copy – 1/n
Probability for 2 copy – 1/n
- - - - - - -
Probability for n-1 copy – 1/n

Step count for 1 copy – 1
Step count for 2 copy – 2
- - - - - - -
Step count for n-1 copy – n-1

Average Complexity = $\sum_{i \,\varepsilon D n} P(i) \times T(i)$

$= (1/n) + (2/n) + \ldots + (n-1)/n = (n-1)/2.$

$((n-1)/2) + 1$ steps.

Compare Algorithms

- Analysis to get an idea about the fastness
- Exact step count is not required
- Example
- An algorithm with step count 10 n+10 < 100n+10.
- Constant associated is not much relevant
  - 100n but it is 40 or 80n.
- $c_3n < c_1n^2+c_2n$

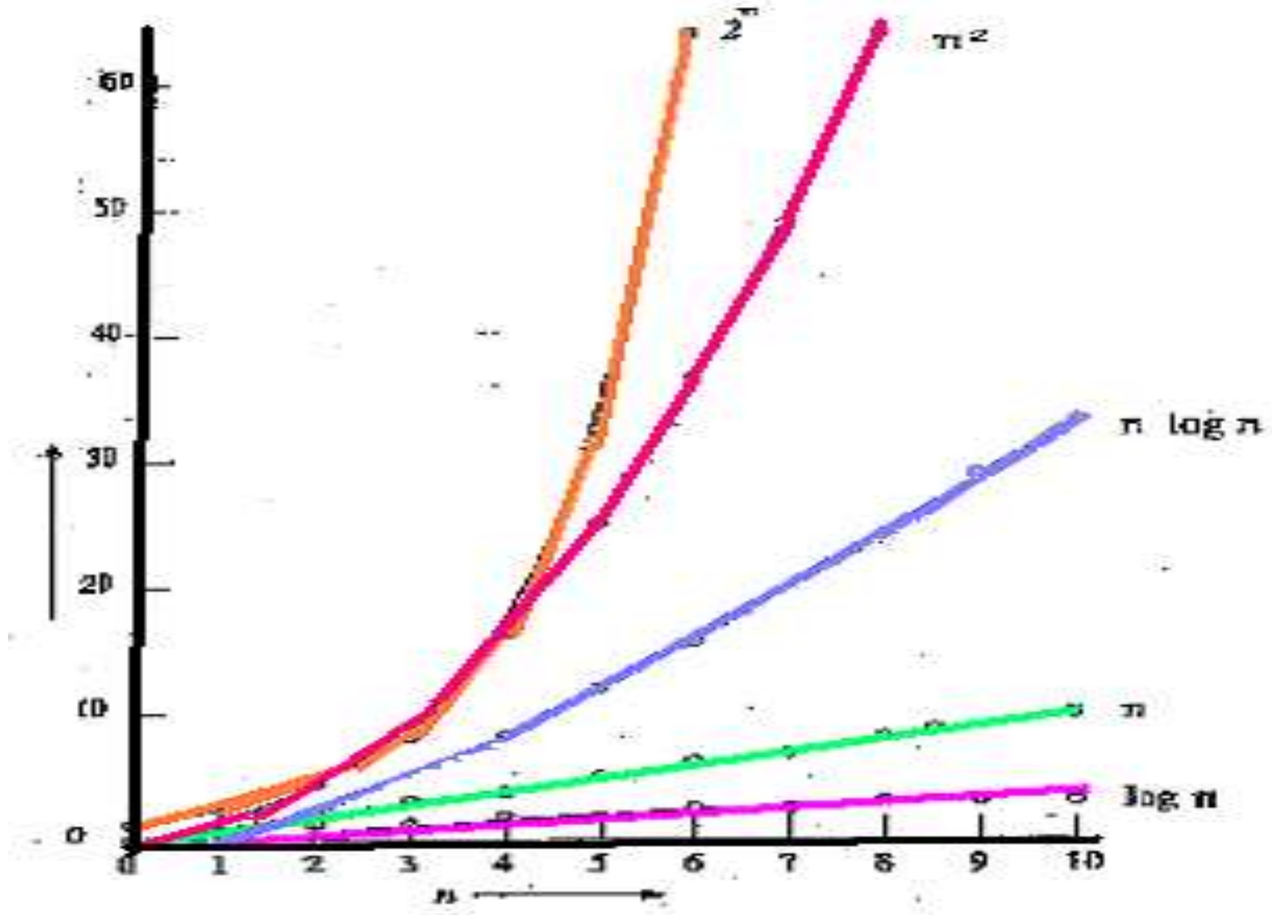
Break Even Point

- Normally $c_3n < c_1n^2+c_2n$
- Irrelevant of c_1, c_2 and c_3 there will be a value of n beyond which c_3n will run faster.
- Exact value of break even point by running
- Sufficient condition for existence
- Compare $c_1n^2+c_2n$ & c_3n
  - break even point exist irrelevant of c_1, c_2 and c_3

Algorithm Analysis

- Precise mathematical analysis is difficult
- Steps to simplify the analysis
  - Identify fastest growing term
  - Neglect the slow growing terms
  - Neglect the constant factor in the fastest growing term are performed.
- Simplification result
  - algorithm’s time complexity.
- Focuses on the growth rate of the algorithm with respect to the problem size

Running Times- Comparison



Asymptotic Notations

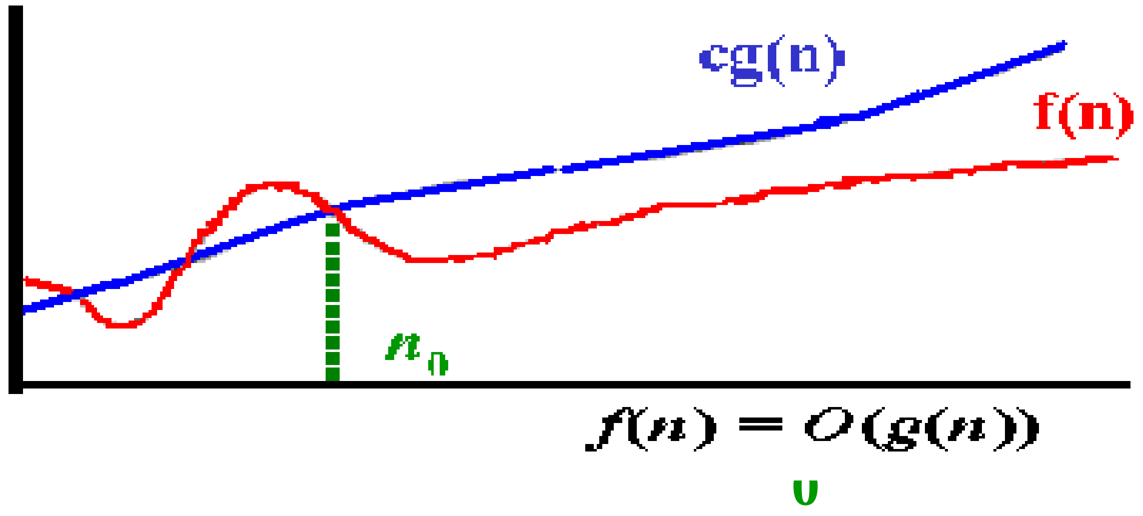
- Describing complexity

Upper Bound

- A set of numbers ranging from a to b.
- Upper bound for the set
- Any number greater than or equal to b
  - Could be b or b+1, or b+2, …
- Moves closer as the number increases
- No point of time it is greater than upper bound.

O-Notation (Upper Bound)

- $f(n) = O(g(n))$ if there are positive constants n_0 and c such that to the right of n_0, the value of $f(n)$ always lies on or below $cg(n)$ or f(n)<=c*g(n) for all n> n_0.



O-Notation Contd..

- If f(n)=O(g(n)) $\rightarrow$f(n) is at least as good as g(n).
- Informally O(g(n)) denotes the set of all functions with a smaller or same order of growth as g(n).
- n^2 belongs O(n^3).

O-Notation Examples

- $n^2+10n = O(n^2)$
- $5n^3+6 =O(n^3)$
- $3\log n+8 = O(\log n)$
- $n^2+10n = O(n^3)$

General Rules for analysis

- In expressing running times, each elementary step such as an assignment, an addition or an initialization is counted as one unit of time
- Leading constants and lower order terms are ignored

Analysis Rules Loops

- The running time of a for loop, is at most the running time of the statements inside the for loop (including tests) multiplied by the number of iterations
- Nested loops should be analyzed inside out.
- The total running time for a statement inside innermost loop is given by its running time multiplied by the product of the sizes of all for loops

Analysis Rules If

- The running time of an if/else statement is not more than the running time of the test, plus the larger of the running times of statements contained inside then and else conditions .

Properties of O notation.

- Constant factors can be omitted
  - $2n^3+6$ ε $O(n^3)$, 2 and 6 are omitted
- Growth rate of a sum is given by the rate of its fastest growing term.
  - $5n^3+3n+8$ ε $O(n^3)$
- if $f(n)>g(n)$, $g(n)>b(n)$ then $f(n)>b(n)$
  - $O(n^2)>O(n)$, $O(n)>O(\log n)$so $O(n^2)>O(\log n)$
- Higher powers of n grows faster than lower powers
  - $O(n^3)> O(n^2)$

Example Problem 1

```
Read(A)
X:= A*A
- - - - - - -
Write(X)
```

any segment for which each statement is executed only once will have a total number of statements executed that is independent of n.

Time complexity will be O(1)

Example Problem2

```
for i :=1 to n do
for j:=1 to n do
{
    ---
}
```

- complexity n*n =O(n^2)

Example Problem3

```
for i:=1 to n do
for j:=1 to i do
{
    ---
}
```

complexity is 1+2+3+4+ -- --+n=n*(n+1)/2=O(n^2)

Example Problem4

```
i=1;
While (i<=n) do
{    ---
    i:= 2*i;
     ---       }
```

- Values of i=1, 2,4, ---,2^k, 2^{k+1} if 2^k <=n <2^{k+1}
- If n is a proper power of 2
  - Loop executed k+1 times where k= $\log_2 n$
- if n is not a proper power of 2
  - Loop executed k+1 times where k = $\lfloor \log_2 n \rfloor$
- Complexity is 1+ $\lfloor \log_2 n \rfloor$ so Θ(logn)

Example Problem2

```
Algorithm power(b,n)
{p:=1; q:=b;
While (n>0) do
{If n is odd, then p:=p*q;
n:=n /2;
q:=q*q;}
Return (p); }
```

- Complexity = O(logn)

Example Problem2

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

- Time complexity as a recurrence relation
- T(n)=a for n=1
- T(n)=T(n-1)+c for n>1
  - Where a and c are contstants

Solving Recurrences

- 3 methods
  – using iteration
  – using recurrence trees
  – using master theorem

Method of substitution / Iteration

- Method expanding recurrence to see a pattern.
- Expand (iterate) the recurrence and express it as a summation of terms dependent only on *n* and the initial conditions

Iteration Method Contd..

- Key focus on 2 parameters
  - the number of times the recurrence needs to be iterated to reach the boundary condition
  - the sum of terms arising from each level of the iteration process
- Techniques for evaluating summations can then be used to provide bounds on solution

Example Problem1

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

$$\begin{aligned} &= T(n-2) + c + c \\ &= T(n-(n-1) + (n-1)\,c \\ &= T(1) + (n-1)\,c \\ &= a + c(n-1) \\ &= O(n) \end{aligned}$$

Example Problem2

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

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

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

- - - - - - - - - - - - - - -

$= \quad T(n-(n-1) + 2+3+. . .+n-1+ n$

$= \quad T(1) + 2+3+. . .+n-1+ n$

$= \quad n*(n+1)/2$

$= \quad O(n^2)$

Example Problem3

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

$= T(n/4)+1+1$

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

- - - - - - - - - - -

$= T(n/2^k)+1+1+1$--- k times

$= T(1)+k$ where $k = \log_2(n)$

$= \log n+1$

$= O(\log n)$

Example Problem4

- $T(1) = 0$
- $T(n) = T(n/2) + n$

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

- - - - - - - - - - -

$= T(n/2^k)+2+4+---+n/2+n$

$= T(1)+ (2^1)+ (2^2)+---+ (2^{k-1})+ (2^k)$

where $k = \log_2(n)$

$= 0+2*(2^k-1)/(2-1)$

$= 2*(n-1)$

$= O(n)$

Example Problem5

- Example of algorithm that ------ into 2 values and seeks to solve both
- $T(n) = 2\ T(n/2) + an$

$$
\begin{aligned}
&= 2^2\ T(n/4) + an + an \\
&= 2^k\ T(n/2^k) + akn \\
&= n + an \log n \\
&= O(n \log n)
\end{aligned}
$$

Example Problem6

- $T(n) = 2\,T(\sqrt{n}) + \log n$ for $n > 2$

Substitute $m = \log n$

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

when $n=2$ $m=1$ Termination condn is $m=1$

Substitute $T(2^m)$ with $S(m)$

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

$= 2^2 S(m/4) + 2*m$

$= 2^k S(1) + k*m$ where $k=\log m$

$= m + m \log m$

$= O(\log n * \log(\log n))$

Example Problem7

Show that

$T(n) = T(\lceil n/2 \rceil) + 1$ is $O(\log n)$

$$
\begin{aligned}
T(n) &= T(n/2) + 1 \\
&= T(n/4) + 1 + 1 \\
&= T(n/2^k) + k \qquad \text{where } k=\log_2 n \\
&= a + k \\
&= a + \log n \\
&= O(\log n)
\end{aligned}
$$

Example Problem8

- $T(n) = 3T(\lfloor n/4 \rfloor) + n$

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

$= n + 3[(n/4) + 3T(n/4^2)]$

$= n[1 + 3/4 + 3^2/4^2] + 3^3 T(n/4^3)$

$= n[1 + 3/4 + 3^2/4^2 + \ldots\ldots + 3^{k-1}/4^{k-1}] + 3^k T(n/4^k)$ where $k = \log_4 n$

$= n[(1*(3/4)^k - 1)/3/4-1] + 3^k a$

$= c * n(3/4)^{\log_4 n} + 4n + 3^{\log_4 n} * a$

$= c * n * n^{\log_4 3/4} + a*n^{\log_4 3} + b*n$

$= c * n^{\log_4 4 + \log_4 3 - \log_4 4} + a\, n^{\log_4 3} + b*n = O(n^{\log_4 3}) + O(n)$

$= O(n)$

Recurrence Tree

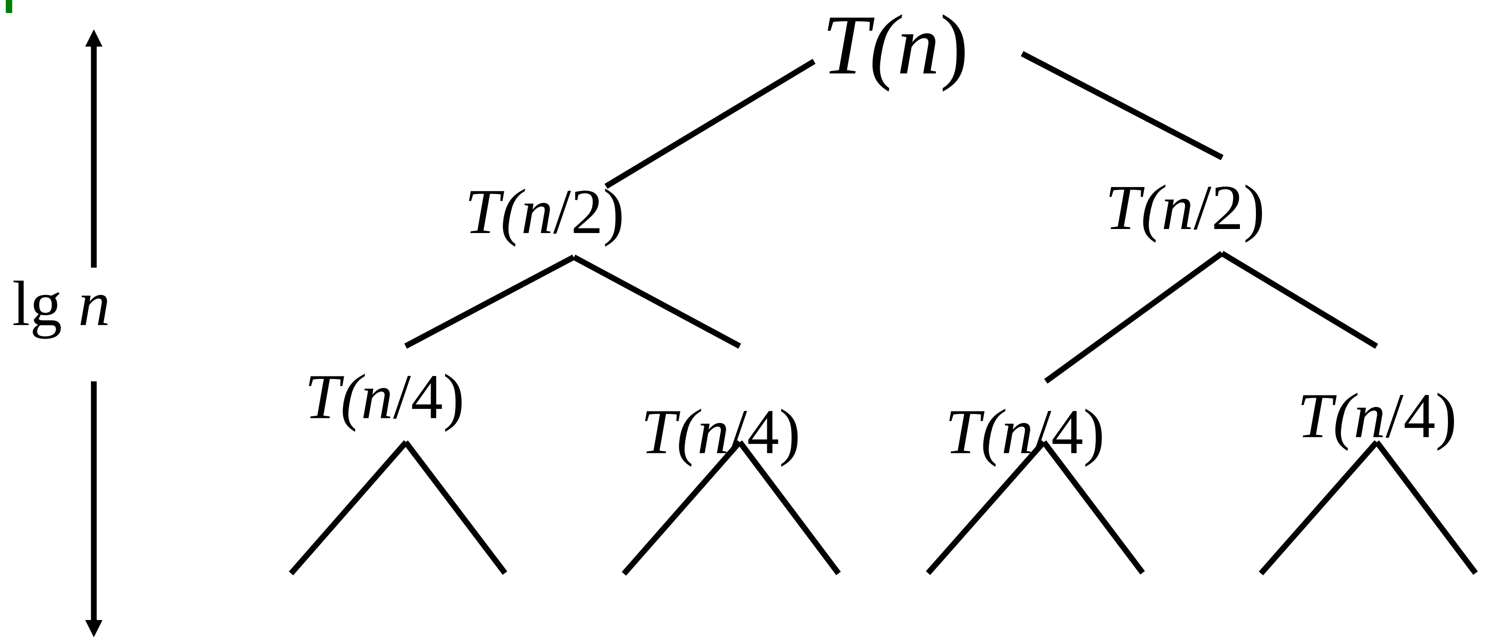
- A convenient way to visualize what happens when a recursion is iterated
- It is good for generating guesses for the substitution method.
- We may describe the execution of a recursive program on a given input by a rooted tree, which is known as recurrence tree

The steps involved in building the Recurrence Tree

- Determine height of tree and the size(sum) of each level as a function of input size
- Add the sizes of each level and

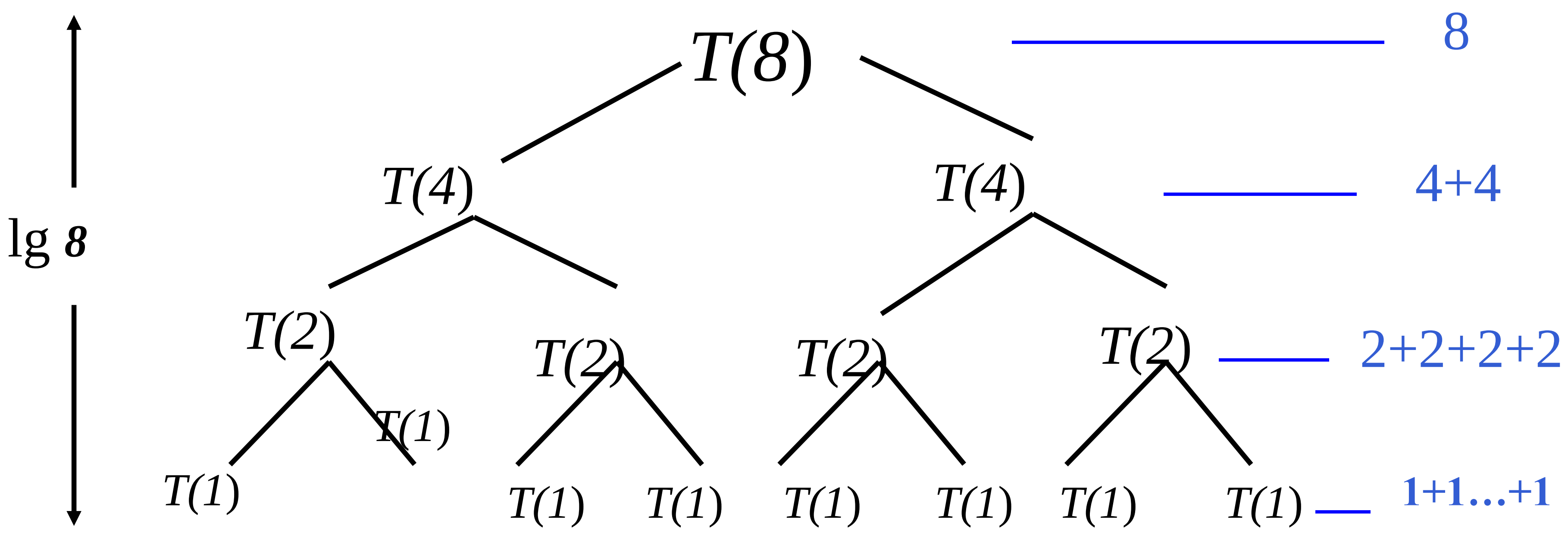
Multiply by the height of the tree

Recurrence Tree Example

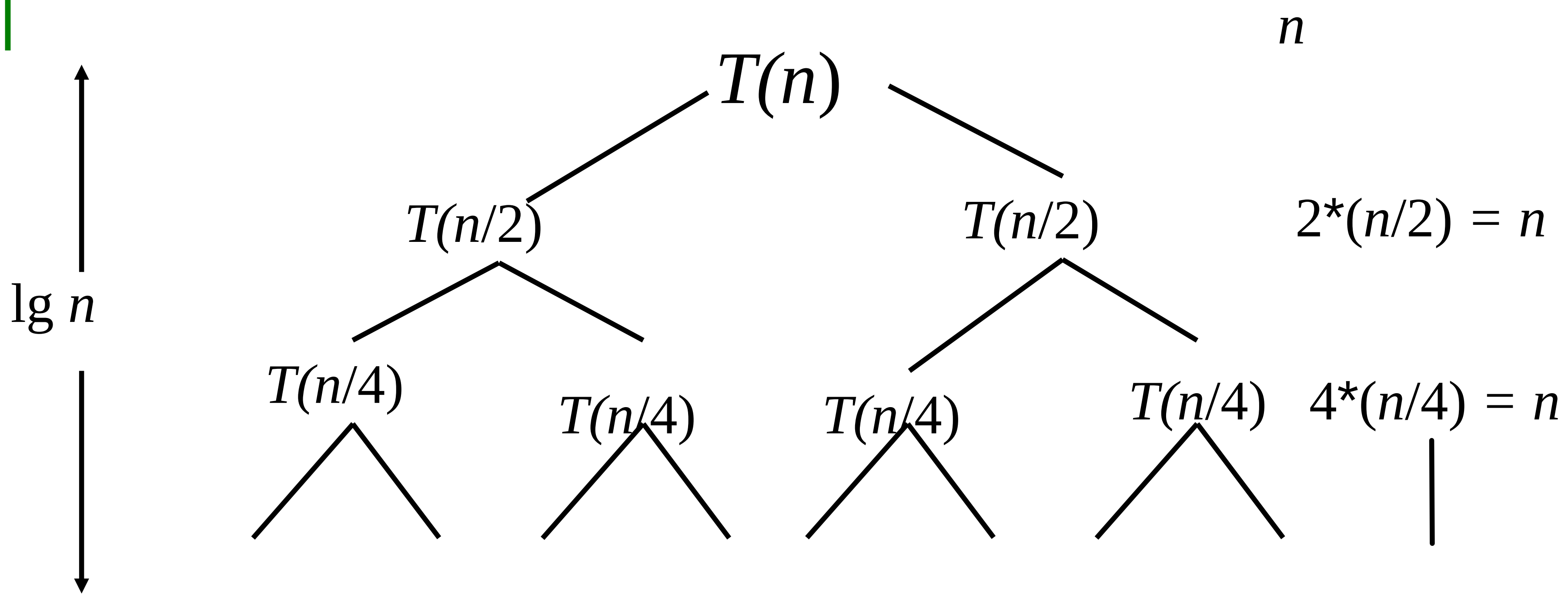


Recurrence Tree Example

- Tree corresponding to T(n)=2*T(n/2)+n for n=8



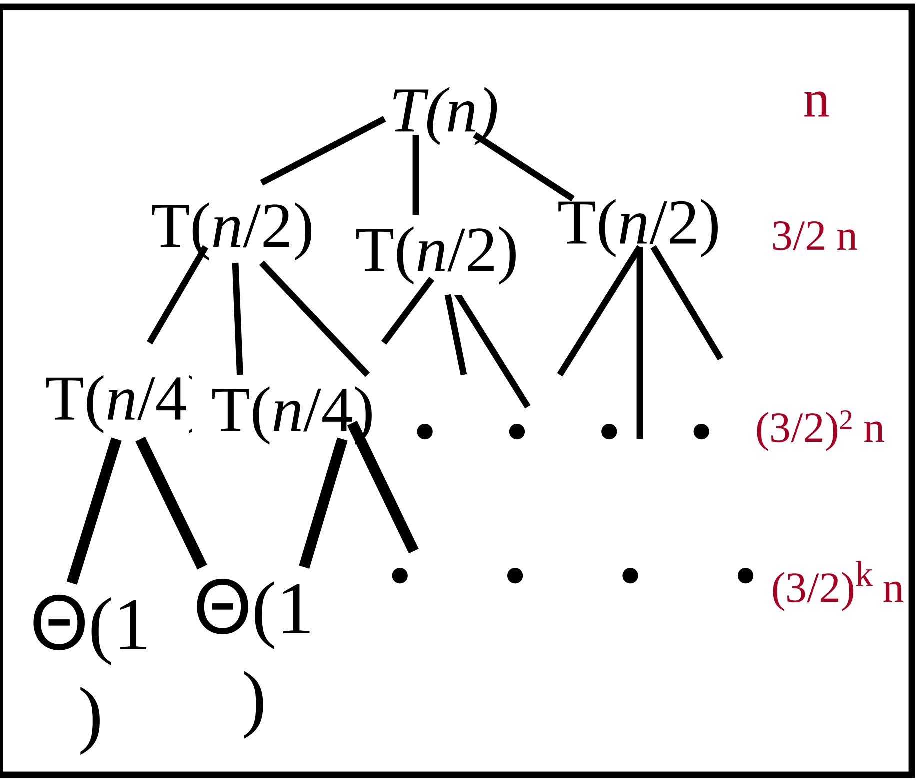
Recurrence Tree T(n)=2*T(n/2)+n



Total: $n \lg n$

Example Problem

- $T(n) \quad = \quad 3T(n/2) + n$



$= n^* [1 + 3/2 + \ldots\ldots\ldots\ldots + (3/2)^k]$

$= n^* [1-(3/2)^{k+1}] / [(3/2)-1]$

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

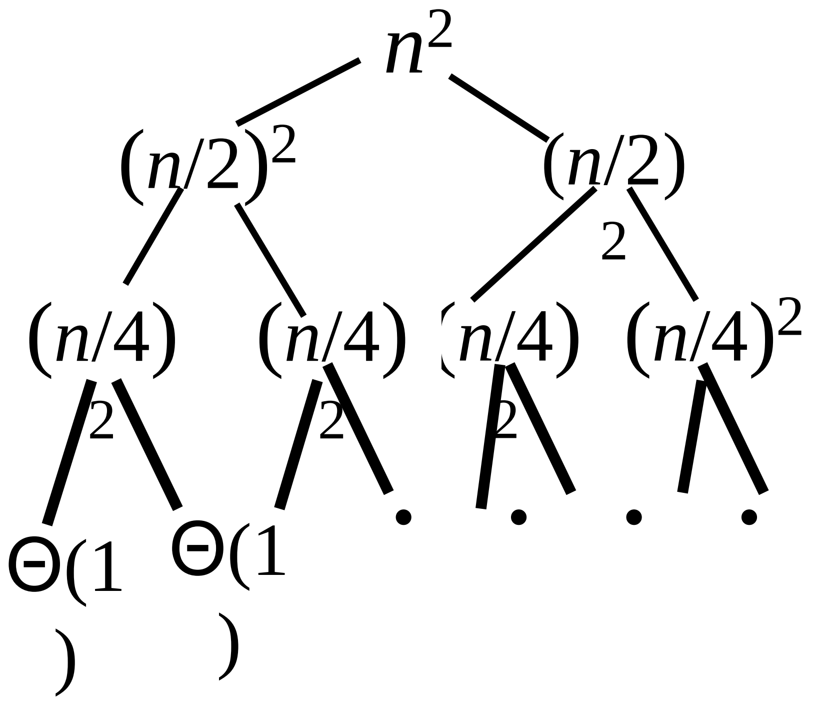
$= 3n *(3/2)^{\log_2 n}$

$= 3n *(3)^{\log_2 n} / 2^{\log_2 n}$

$= 3 *(n)^{\log_2 (3)} = O(n^{1.6})$

Example Problem

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



$n^2 + n^2/2 + n^2/4 + \ldots\ldots\ldots\ldots\ldots$

$= n^2 [1 + ½ + \ldots\ldots\ldots\ldots (1/2^k)]$

$= n^2 [1-(1/2)^{k+1}] /(1/2)$

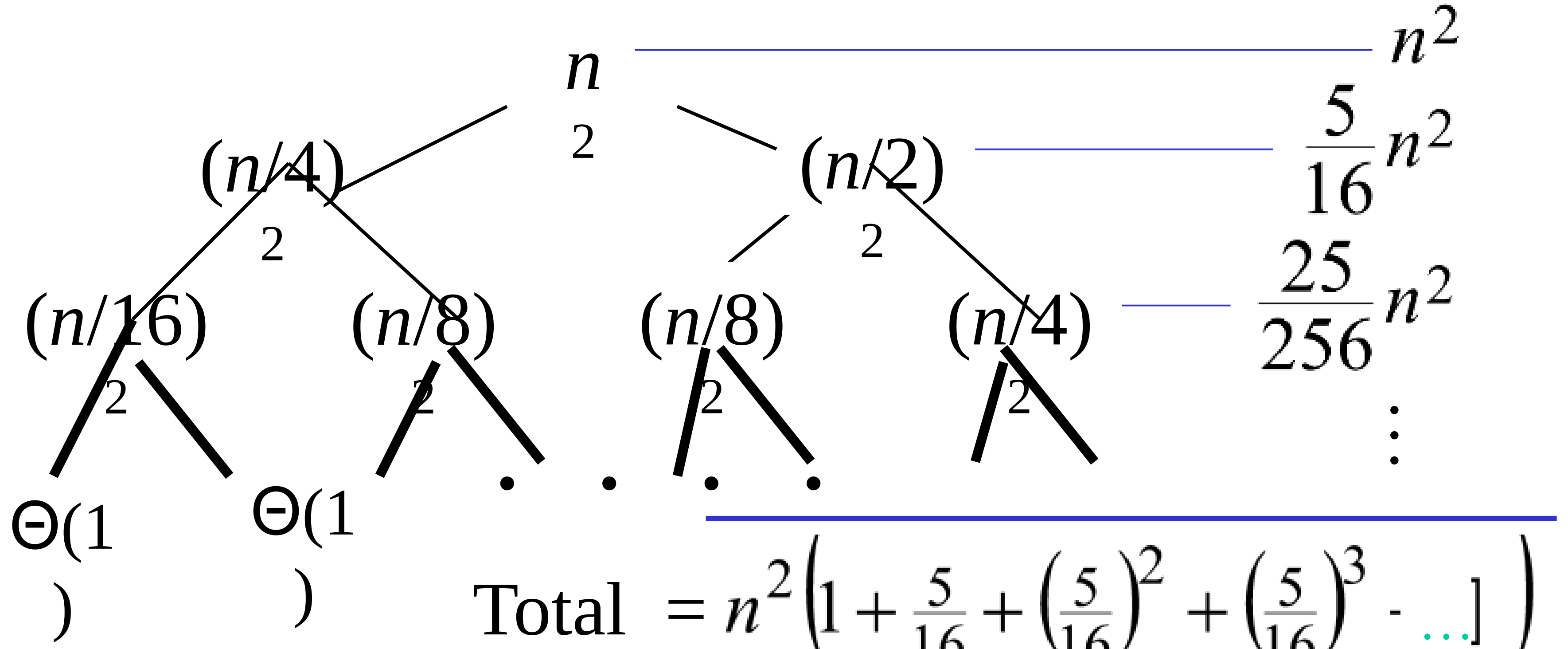
$= 2n^2[1-(1/2^{k+1})]$

$= 2 n^2[1-1/2n]$

$= O(n^2)$

Example Problem

Solve $T(n) = T(n/4) + T(n/2) + n^2$

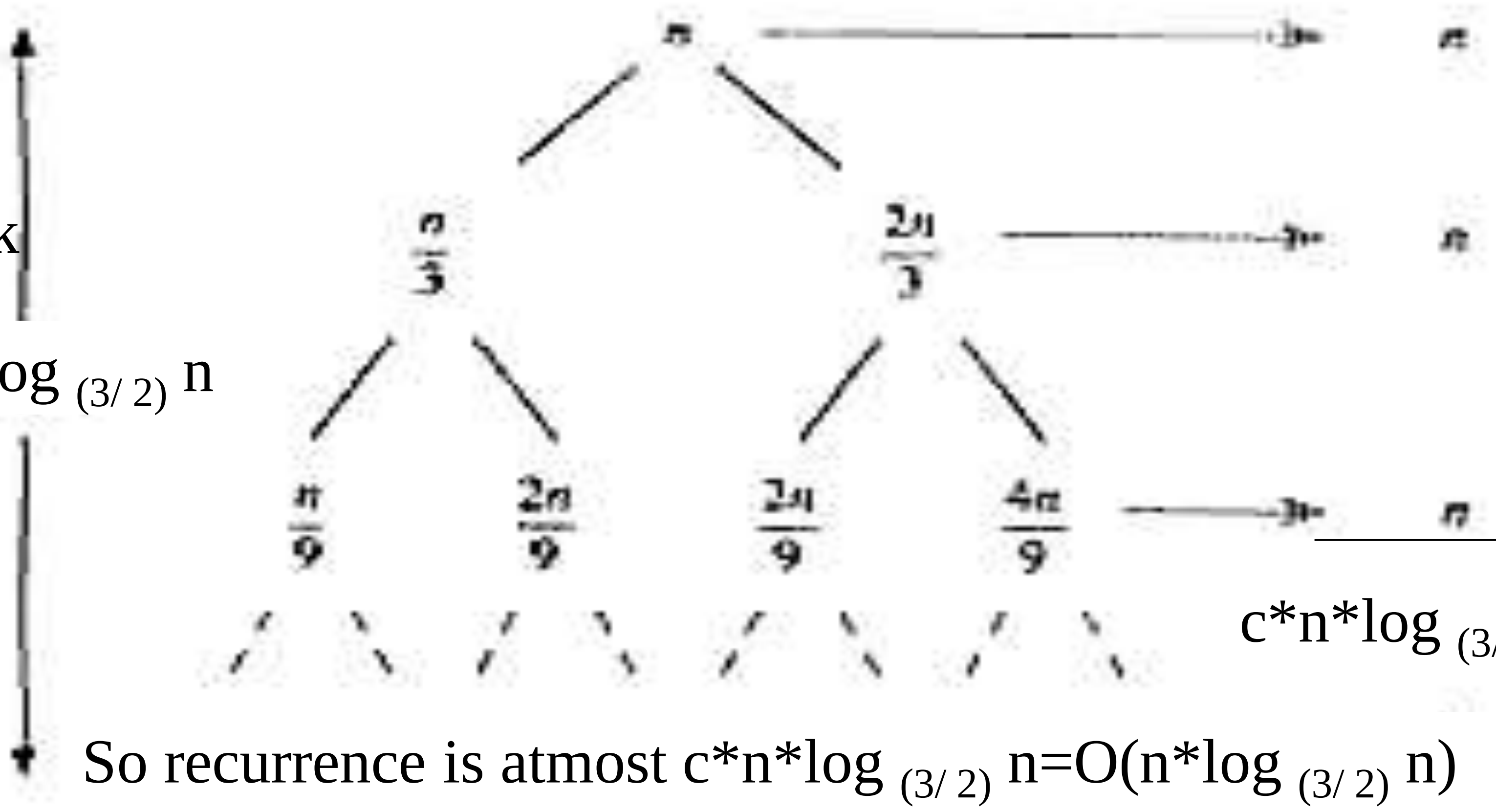


$$\text{Total} = n^2\left(1 + \frac{5}{16} + \left(\frac{5}{16}\right)^2 + \left(\frac{5}{16}\right)^3 + \dots\right)$$

$$= \Theta(n^2)$$

Example Problem

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



$n=(3/2)^k$

So $k = \log_{(3/2)} n$

$c*n*\log_{(3/2)} n$

So recurrence is atmost $c*n*\log_{(3/2)} n=O(n*\log_{(3/2)} n)$

Master Theorem

$T(1) = d$

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

Has solution

$T(n) = O(n)$ if $a<b$

$T(n) = O(n\log n)$ If $a=b$

$T(n) = O(n^{\log_b a})$ if $a>b$

Verification Example 1

$T(1)= 0$

$T(n)=4T(n/2)+cn$ for $n=2$

$a = 4$; $b = 2$; $a>b$

$T(n) = O(n^{\log_b a})$ if $a>b$

$T(n) = O(n^{\log_2 4}) = O(n^2)$

Verification Example 2

$T(1)= 0$

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

$a = 3; b = 2; a>b$

$T(n) = O(n^{\log_b a})$ if $a>b$

$T(n) = O(n^{\log_2 3}) = O(n^{1.6})$

Verification Example 3

$T(1) = 0$

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

$a = 1;\ b = 2;\ a<b$

$T(n) = O(n)$ if $a<b$

$T(n) = O(n)$

Verification Example 4

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

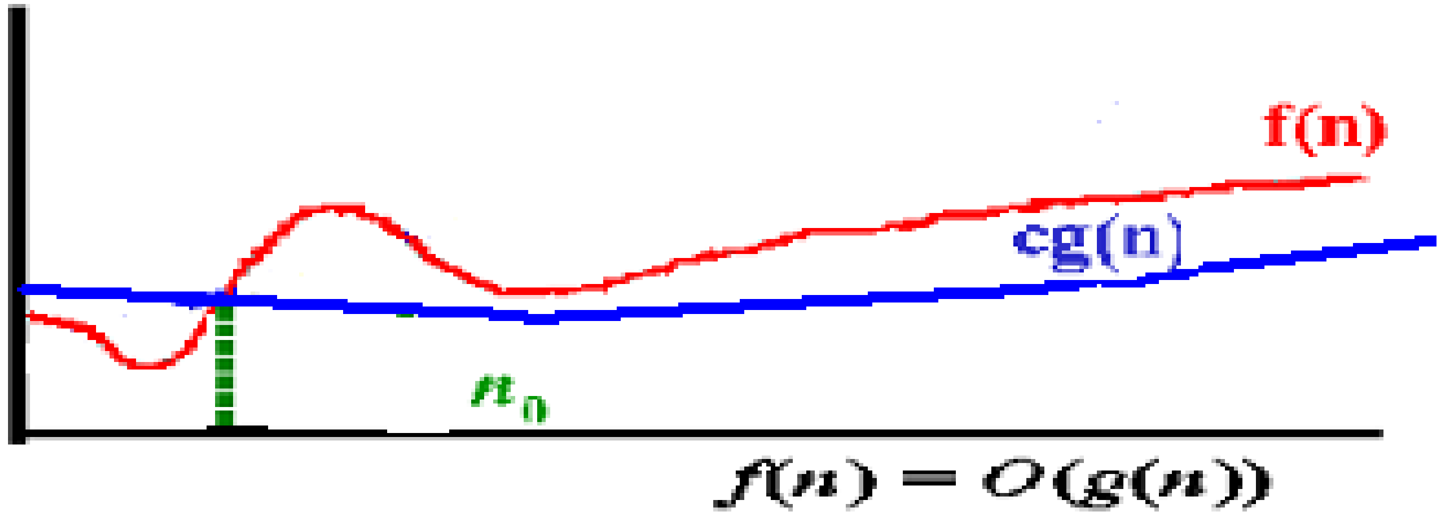
a = 2; b = 2; a=b

$T(n) = O(n\log n)$ If a=b

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

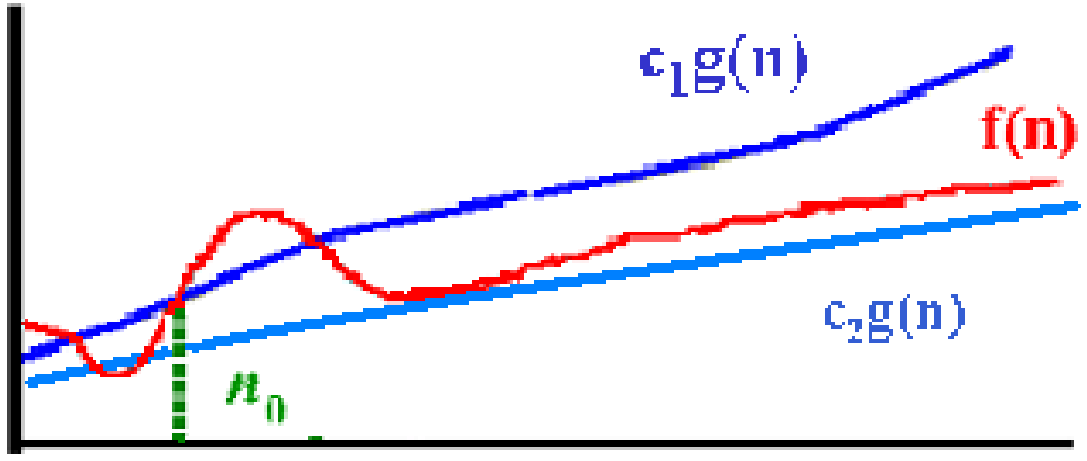
Ω-Notation (Lower Bound)

- $f(n) = \Omega(g(n))$ if there are positive constants n_0 and c such that to the right of n_0, the value of $f(n)$ always lies on or above $cg(n)$.



θ -Notation

- $f(n) = \theta(g(n))$ if there exist positive constants n_0, c_1 and c_2 such that to the right of n_0 the value of $f(n)$ always lies between $c_1g(n)$ and $c_2g(n)$ inclusive.



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

Other notations

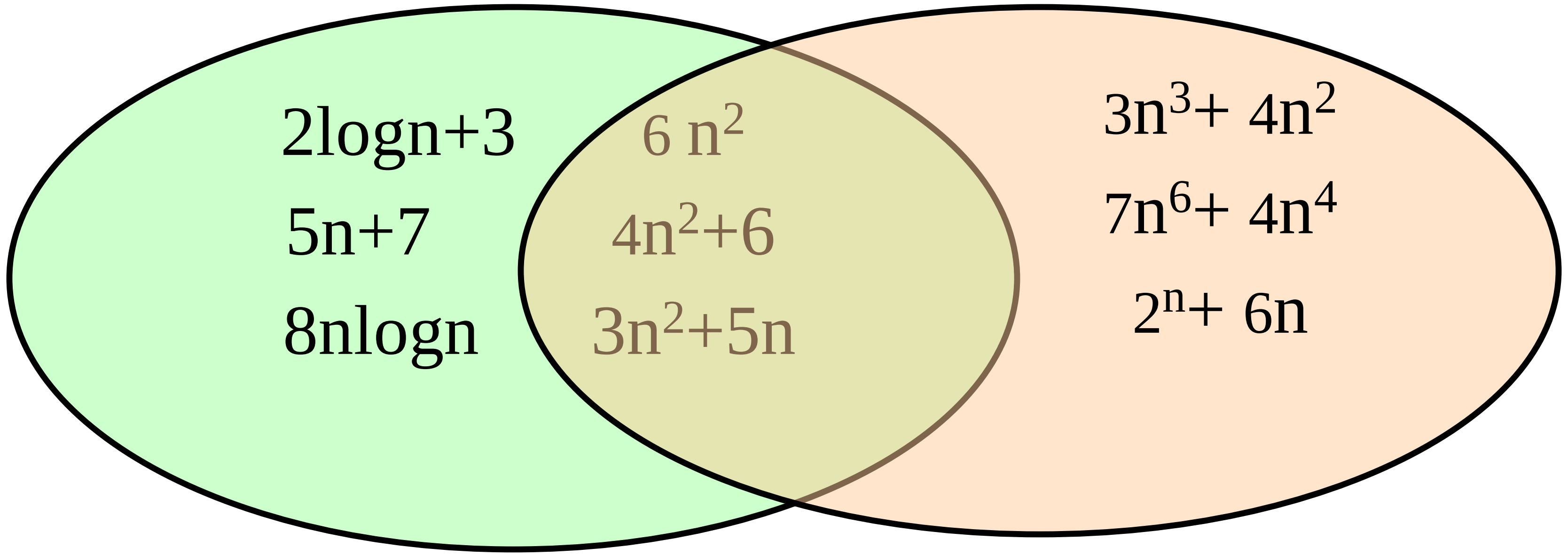
- for every positive real constant c there exists a nonnegative integer N such that for all n>= N
- $f(n)<=c*g(n)$
- so 5n is $o(n^2)$ 6 log(n) is o(n) and 8n is o(nlogn)
- The asymptotic upper bound provided by O may or may not be asymptotically tight. But o small is used to denote an upeer bound that is not asymptotically tight.

Other notations Contd..

- Similarly ω(small omega) notation is used to denote a lower bound that is not asymptotically tight

Examples

$O(n^2)$ $\theta\ (n^2)$ $\Omega\ (n^2)$



Examples Contd..

$o(n^2) = O(n^2) - \boldsymbol{\theta}(n^2)$

$2\log n+3$	$6n^2$
$5n+7$	$4n^2+6$
$8n\log n$	$3n^2+5n$

General Master Theorem

$T(1) = \theta(1)$

$T(n) = aT(n/b) + \theta(n^\alpha)$

Has solution

$T(n) = \theta(n^\alpha)$ if $\alpha \bullet \beta$

$T(n) = \theta(n^\alpha)\log_b n$ if $\alpha = \beta$

$T(n) = \theta(n^\beta)$ if $\alpha \bullet \beta$

Posterior Analysis

- Technique of coding a given solution and then measuring its efficiency.
- Provides the actual time taken by the program.
- Draw back
  - Depends upon the programming language, the processor and a lot of other external parameters.

Comparison

- Gettime() is a function that returns the current time in milli seconds.
- The issues
  - What value of n to be selected
  - Which data set is to be used
    - best,average or worst case
  - What is the accuracy required.
  - How many values of n are required.

Comparison Contd..

- If asymptotic behaviour is already known then 2 or 3 values can generate the curve.
- Asymptotic behaviour omits initial values of n and constant terms involved
  - For an accurate estimate more values of n
  - More samples needed from small vales of n.

Example Case

- A reasonable set of values for n   for the sequential search algorithm is

  10, 20, 30, 40, 50, 60, 70, 80, 90,

  100, 200, 300, 400, 500, ----- .

Example Algorithm

```
Algorithm seqsearch(a,x,n)
{
i:=n; a[0]:=x;
while (a[i]<>x) do i:=i-1;
return i;
}
```

worst case when x is which is not present in a.
for definiteness set a[i]=i for 1 to n and x=0

Profiling Example- Search

- For definiteness element searched is taken as 0
- Search is performed with different array sizes
- Search is repeated many times for more accuracy
- array n stores different sizes of a for search

Algorithm for Profiling

```
Algorithm timesearch()
{ for j:=1 to 1000 do a[j]:=j;
for j:=1 to 10 do  /*array n stores different sizes of a for search */
{ n[j]:= 10*(j-1);
 n[j+10]:=100*j; }  /* generating different n*/
for j:=1 to 20 do
{ h:= gettime();
k:=seqsearch(a,0,n[j]); /* x is taken as 0*/
h1:=gettime();
t:=h1-h;
write(n[j],t); } }
```

Worst Case analysis

- For smaller values of n , repetition factor should be large.
- Generating data
  - Difficult to find all possible cases and its maximum
  - Usually available maximum is chosen as the worst case.

Average Case analysis

- More difficult than worst case.
- For n inputs there will n! combinations
- Determining all cases and their average is a very difficult task.
- Average of a limited subset is usually determined.

Assignment 1 Problem1

Solve $T(n) = 3T(\lfloor n/4 \rfloor) + n$

Assignment 1 Problem2

Solve

$$T(n) = \begin{cases} a & \text{for } n<=2 \\ 8\,T(n/2)+bn^2 & \text{for } n>2 \end{cases}$$

Assignment 1 Problem2

Solve

$$T(n) = \begin{cases} a & \text{for } n<=2 \\ 8\,T(n/2)+bn^2 & \text{for } n>2 \end{cases}$$

Example Problem1

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

$$
\begin{aligned}
&= T(n-2) + c + c \\
&= T(n-(n-1) + (n-1)\,c \\
&= T(1) + (n-1)\,c \\
&= a + c(n-1) \\
&= O(n)
\end{aligned}
$$

Example Problem1

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

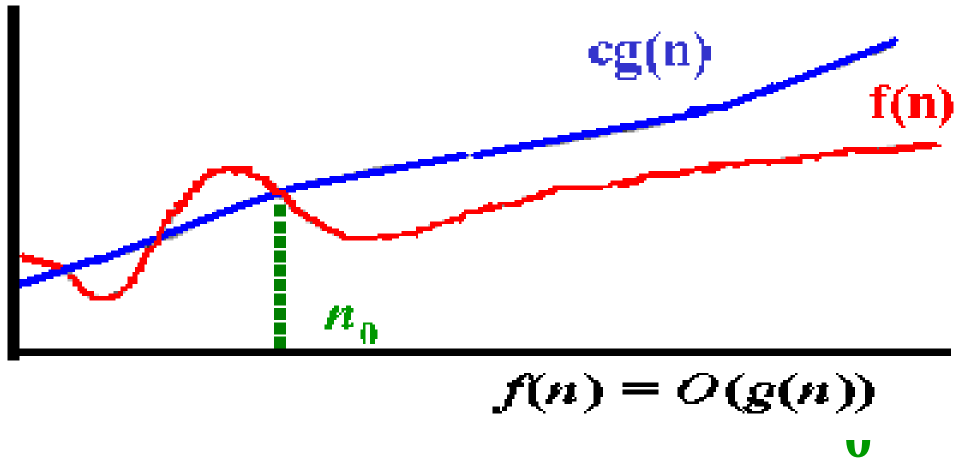
$$\begin{aligned} &= T(n-2) + c + c \\ &= T(n-(n-1) + (n-1) c \\ &= T(1) + (n-1) c \\ &= a + c(n-1) \\ &= O(n) \end{aligned}$$

O-Notation (Upper Bound)

- $f(n) = O(g(n))$ if there are positive constants n_0 and c such that to the right of n_0, the value of $f(n)$ always lies on or below $cg(n)$ or f(n)<=c*g(n) for all n> n_0.

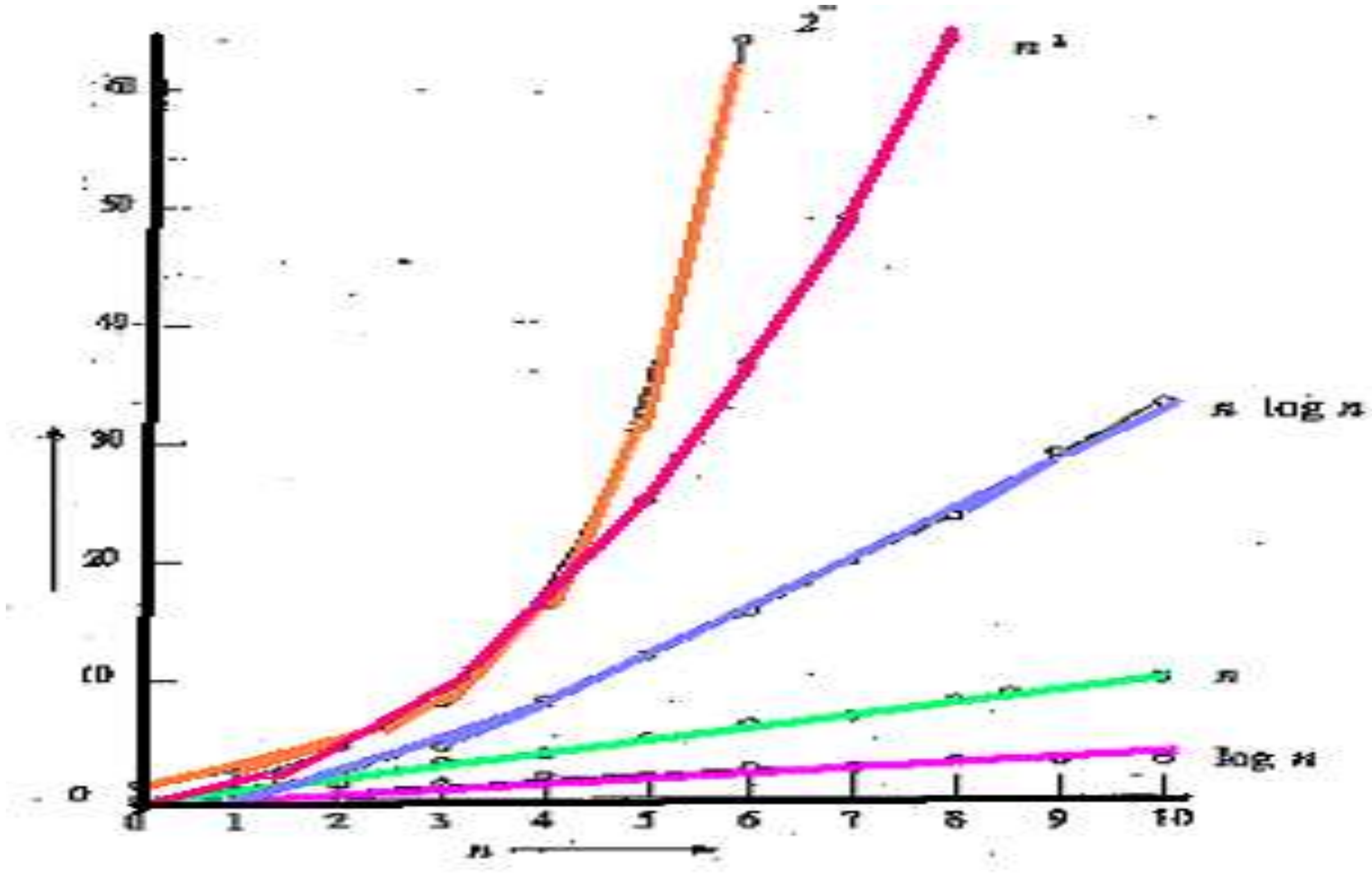
cg(n)

f(n)



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

Running Times- Comparison

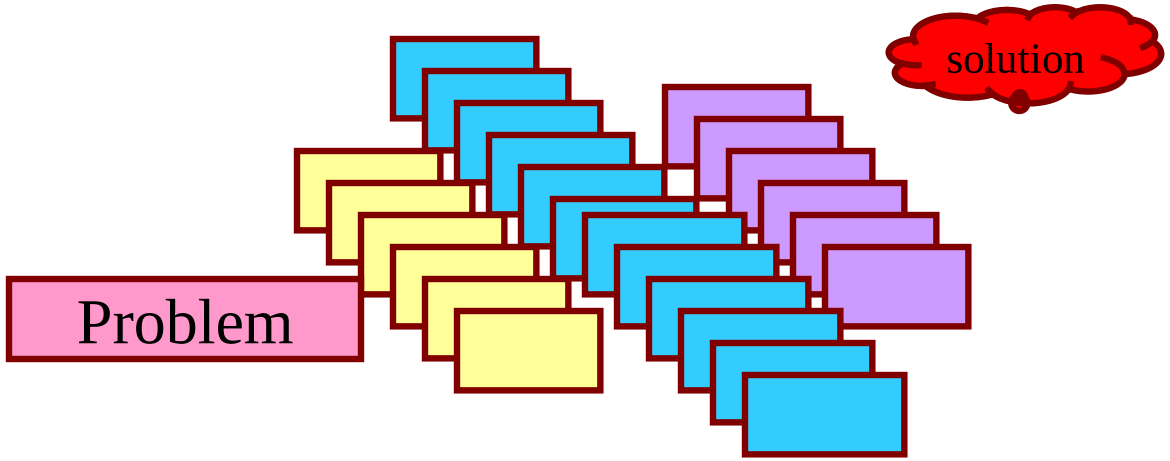


ALGORITHM ANALYSIS AND DESIGN

DIVIDE AND CONQUER

Strategy

- Given a function on n inputs
  - input splitted into k disjoint subsets
  - yielding k subproblems.
- Solve subproblems
- Combine the subsolutions into a solution



Solving Subproblems

- Large Subproblems
  - Solved by reapplication of divide and conquer.
  - Subproblems same type as the original problem implemented using recursive algorithm
- Smaller subproblems solved independently.

Control Abstraction

- Procedure whose flow of control is clear.
- Primary operations are specified by other procedures whose precise meanings are left undefined

Control Abstraction for DandC

```
Algorithm DandC(P)
{ if small(P) then return(P);
else
{divide P into smaller instances P1,P2,- - -, Pk for k>1
apply DandC to each subproblem ;
return combine(DandC(P1),- - - ,DandC(Pk));}}
```

Time Complexity

$$T(n) = \begin{cases} g(n) \text{ when n is small} \\ T(n1)+T(n2)+\text{- - -}+T(nk)+f(n) \end{cases}$$

- when subproblems are similar, complexity can be represented by a recurrence

$$T(n) = \begin{cases} T(1) & n=1 \\ a^* \, T(n/b) + f(n) & n>1 \end{cases}$$

Time Complexity Contd..

- In general
- $T(n)=n^{\log_b a}[t(1)+u(n)]$ where $u(n)= {}_{j=1}\varepsilon^{k} h(bj)$
- $H(n)=f(n)/ n^{\log_b a}$

- If $h(n)$ is $O(n^r)$ for $r>0$ then $u(n)$ is $O(1)$
- If $h(n)$ is $\theta(\log n^r)$ for $r=>0$ then $u(n)$ is $\theta(\log n^{r+1} / (r+1))$
- If $h(n)$ is $\Omega(n^r)$ for $r>0$ then $u(n)$ is $\theta(h(n))$

Example

$$T(n) = \begin{cases} 1 & \text{for } n=1 \\ T(n/2)+c & \text{for } n>1 \end{cases}$$

$h(n)=f(n)/n^{\log_b a}=c*(\log n)^0$

So $u(n)= \theta(\log n)$

$T(n)=n^{\log 1}\,[c+\theta(\log n)] = \theta(\log n)$

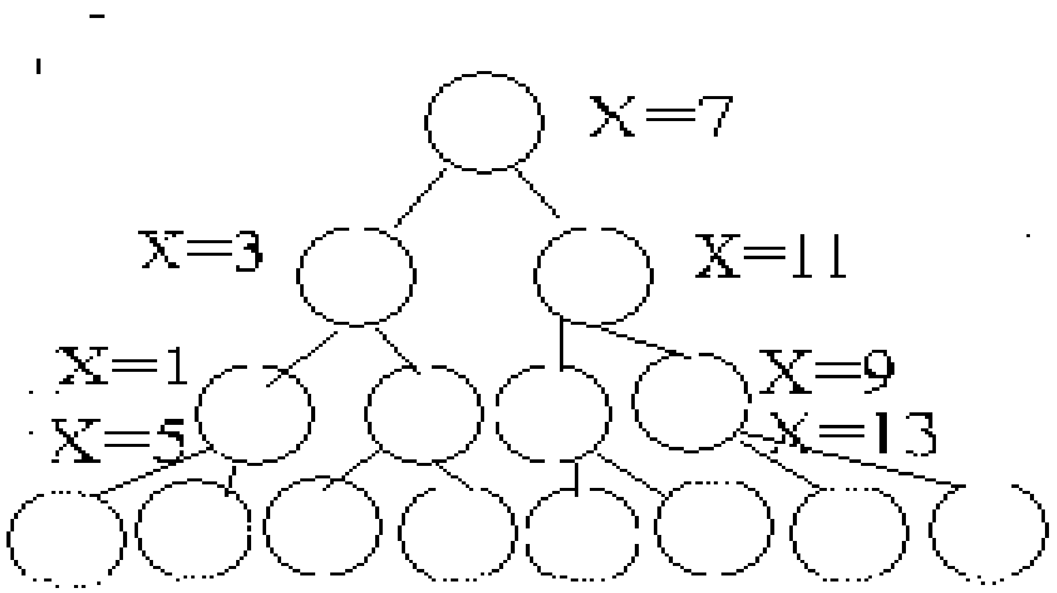
Binary Search

```
Algorithm binsearch(a,i,j,x)
{if (i=j) then
{ if (x=a[i]) then return i;
else return 0; }
else
{mid:= (i+j) div 2;
if (x=a[mid] then return mid;
else if (x<a[mid]) then
return binsearch(a,i,mid-1,x);
else return binsearch(a,mid+1,j,x); }}
```

Computing Complexity

Example

1	3	5	7	9	11	13



Unsuccessfull cases

Binary Search Complexity

- unsuccessful search θ (logn)
- Successful
  - best - θ (1)
  - worst – when leaf node is reached - θ(logn)

Average Case Complexity

$T(n)= 2^0*1+2^{1}*2+- - - + 2^{k-1} * k$

$= [2^1*1+2^{2}*2+- - - + 2^{k-1} *(k-1)] +[2^0+2^{1}+- - - + 2^{k-1}]$

$= [(k-2)* 2^{k-1}+2] + [2^{(k+1)}-1]$

since we have $\sum_{i=1}^{k} i*2^i =(k-1)*2^{(k+1)}+2$

$=k*2^{k}+2$

$=\theta (n\log n)$

so average case complexity $\theta(\log n)$

Finding Max & Min

```
Algorithm smaxmin(i,j,n,max,min)
{max:=a[1]; min:=a[1];
For i:= 2to n do
{if (a[i]>max) then max:=a[i];
if (a[i]<min) then min:=a[i];
}}
```

2(n-1) Comparisons

Improvement

```
Algorithm smaxmin(i,j,n,max,min)
{max:=a[1]; min:=a[1];
For i:= 2to n do
{if (a[i]>max) then max:=a[i];
else if (a[i]<min) then min:=a[i];
}}
```

Worst case- sorted in descending order - 2(n-1) Comparisons

Best case- sorted in ascending order -   n-1Comparisons

Average case – half cases a[i] is greater - n/2+(n-1) =3n/2 -1

Divide and Conquer Approach

- Split input into smaller subsets
- Repeat until input size is 1 or 2

MAXMIN

```
Algorithm maxmin(i,j,max,min)
{if (i=j) then max:=min:=a[i];
else if (i=j-1) then
if (a[i]<a[j]) then
{max:=a[j];min:=a[i];}
else {max:=a[i];min:=a[j];}}
else
{ mid:= L(i+j)/2□  ;
maxmin( i,mid,max,min);
maxmin(mid+1,j,max1,min1);
if (max<max1) then max:=max1;
if (min>min1) then min:= min1;}}
```

Time Complexity

T(n)= T(n/2)+T(n/2)+2 for n>2; =1 for n=2

When n is a power of 2, T(n) = 2*T(n/2) +2

= - - - - - -

$= 2^{k-1} * T(2) + 2 * (2^{k-1} – 1) = 2^{k-1} + 2^{k} – 2$

$= 3*n/2 – 2$

it is the best, average and worst case complexity.

Compared to the 2n-1 comparisons of straight maxmin this approach is better.

But it requires logn +1 levels of stack.

Considering Index Comparisons

- When element and index comparisons of the same cost
- In languages that does not support switch statement modification required

Improvement

```
Algorithm maxmin(i,j,max,min)
{if (i ≥  j) then
if (a[i]<a[j]) then {max:=a[j];min:=a[i];}
else {max:=a[i];min:=a[j];}}
else
{ mid:= L(i+j)/2□  ;
maxmin( i,mid,max,min);
maxmin(mid+1,j,max1,min1);
if max<max1) then max:=max1;
if (min>min1) then min:= min1;}}
```

Complexity

$C(n)= 2* C(n/2)+3$ for $n>2$

$=2$ for $n=2$

- unfolding recurrence
  - $C(n) = 2^{k-1}*C(2) + 3* {}_0\varepsilon^{k-2} 2^i$

    $=2^k +3* 2^{k-1} - 3$

    $=5n/2 - 3$

Comparison

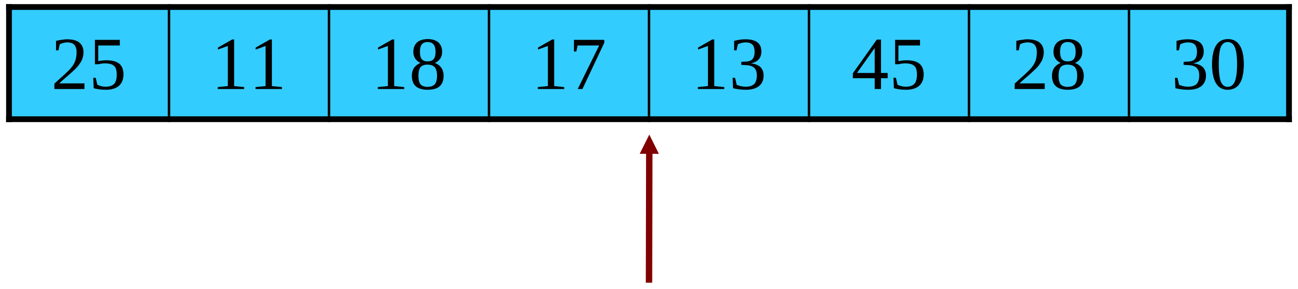
- Better than straight maxmin 3*(n-1)
- Practically slower due the overhead of stacking

Summary

- When element comparisons are costlier dandc yields a better algorithm.
- Dandc always willnot give better algorithm.
  - Only a design technique that will guide to better designs.
- Constants should be specified, during comparisons if relevant(when both has same order complexity).

Merge Sort

- Divides list into two sub lists
- Sort sub lists separately
  - Recursive invocation
- Merges the sub lists



Algorithm Mergesort

```
Algorithm mergesort(low,high)
{ if (low<high) then
   { mid:= ⌊ (low+high)/2⌋
     megesort(low,mid);
     mergesort(mid+1,high);
     mege(low,mid,high);
}}
```

Algorithm Merge

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

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

Complexity

$$T(n) = \begin{cases} 2 \cdot T(n/2) + cn & \text{for } n>1 \\ a & \text{for } n=1 \end{cases}$$

- Unfolding recurrence

$$\begin{aligned} T(n) &= 2\ T(n/2) + cn \\ &= 2^2\ T(n/4) + 2cn \\ &= 2^k\ T(n/2^k) + kcn \\ &= 2^k\ T(1) + kcn = an + cn \log n \\ &= O(n \log n) \end{aligned}$$

Refinements

- 2n locations – Extra n locations
  - Associate an extra field with key
- For small values of n recursion inefficient
- Much time is wasted on stacking.
  - Use an efficient nonrecursive sort for small n

Refinement 1

For n<16 insertion sort is used.

```
Algorithm insertion sort(a,n)
{for j:=2 to n do
{ item := a[j]; i := j-1;
while ((i>=1) and (item<a[j])) do
{a[i+1]:= a[i];   i := i-1; }
a[i+1] := item; } }
```

Complexity

- Insertion sort for worst case
  - $_2$ ε ^{n} j=n(n+1)/2-1=θ(N^2).
- In the best case
  - θ(N).

Algorithm2

```
algorithm mergesort2(low,high)
{if (high-low)<15 then      ///when size is <16
return insertionsort1(a,link,low,high)
else
{  mid := L(low+high)/2 □       ///divides into 2
q:=mergesort(low,mid);
r:=mergesort(mid+1,high);
return merge1(q,r);}}
```

Refinement 2

- An auxillary array with values 0..n used
- Each index points to the original array

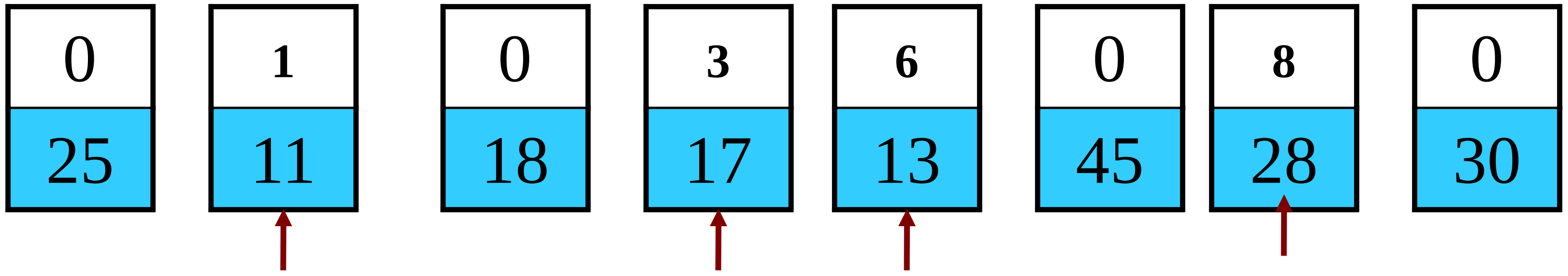
- Interpreted as pointers to elements of a

0	0	0	0	0	0	0	0
25	11	18	17	13	45	28	30

Demonstration

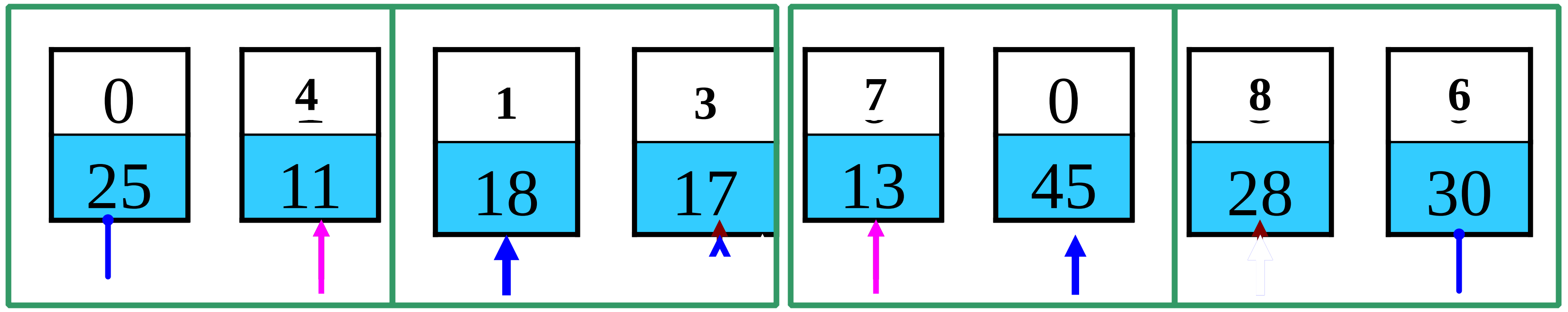
- Interpreted as pointers to elements of a

	1	2	3	4	5	6	7	8
link	0	0	0	0	0	0	0	0
value	25	11	18	17	13	45	28	30



Demonstration Contd..

0	1	0	3	6	0	8	0
25	11	18	17	13	45	28	30
1	2	3	4	5	6	7	8



Demonstration Contd..

0	4	1	3	7	0	8	6
25	11	18	17	13	45	28	30
1	2	3	4	5	6	7	8

7	5	1	3
25	11	18	17

4	0	8	6
13	45	28	30

Sorting Over

Algorithm3

```
Algorithm merge1(q,r)
{i:=q;j:=r;k:=0;
while ((i<>0) and j<>0)) do
{if (a[i] ≤ a[j]) then
{link[k]:=i;k:=i;i:=link[i];}
else {link[k]:=j;k:=j;j:=link[j];}}
If (i = 0) then link[k]:=j; else link[k]:=i;
return link[0];}
```

ALGORITHM ANALYSIS AND DESIGN

GREEDY STRATEGY

Introduction

- Solution to problems with n inputs
- Required to obtain a subset that satisfies some constraints
- Optimization meassure
- Best choice at any moment

Terminology

- Objective function
  - Function representing the optimization measure
- Feasible solution
  - Any subset that satisfies the given constraints
- Optimal solution
  - Feasible solution that either maximizes or minimizes a given objective function.

Working

- Works in stages – One input at a time
- At each stage, decision about a particular input
- Inputs one at a time based on selection procedure
- Discarded if Inclusion results in an infeasible solution

Selection Procedure

- Based on the optimization measure
- Different Optimization measures possible for a problem.

Example

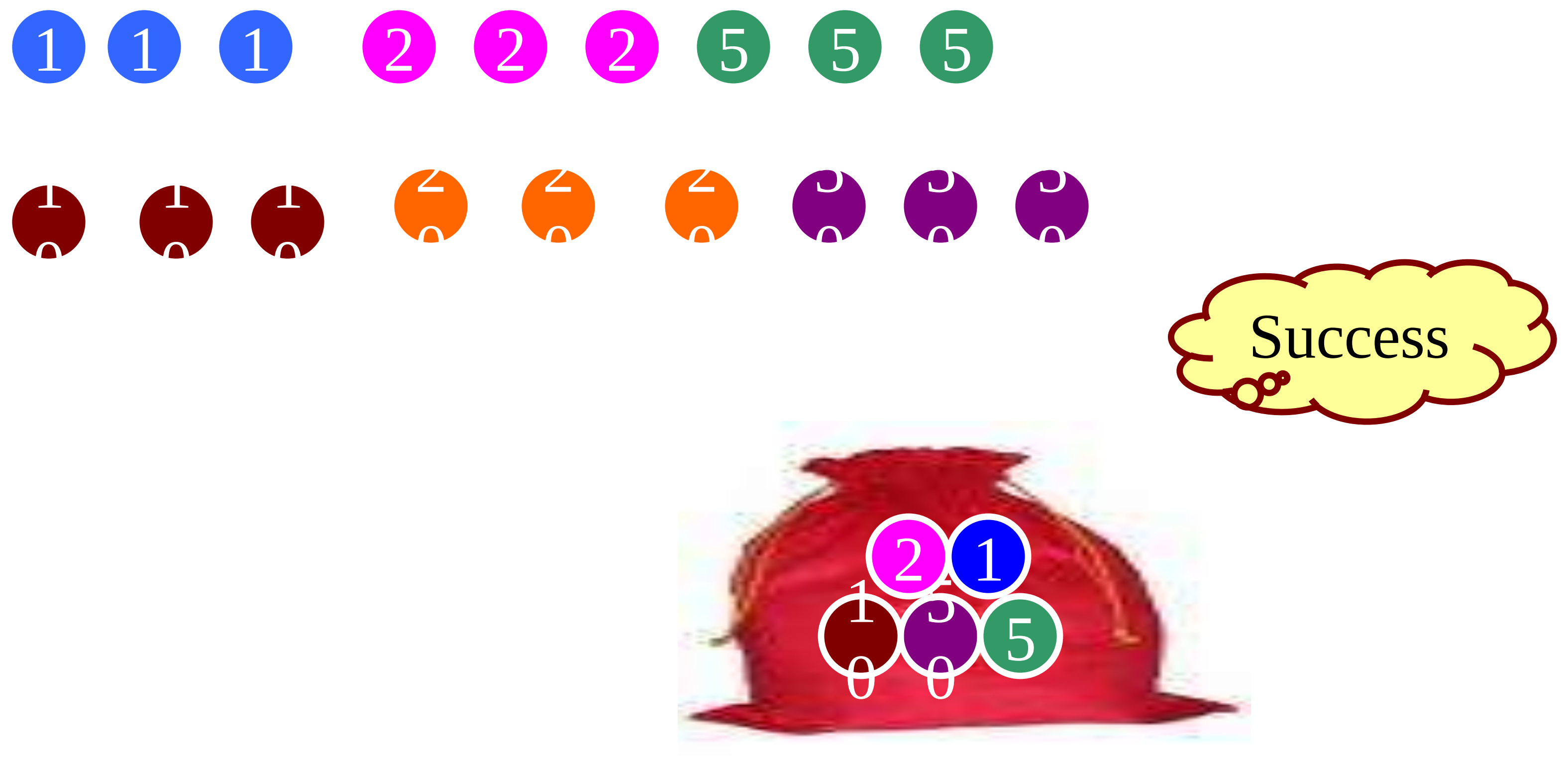
- Taking change a particular amount given a collection of coins
- Minimise the number of coins used
- Example Case
  - Amount to be raised
  - 68 paise
  - Coins available
  - 1 , 1 , 1 ,2,2,2,5,5,5,10,10,10,20,20,20,50,50,50

Greedy Solution

- Selection procedure
  - value of the coin
  - highest one first
- Feasibility of partial solution
  - Sum of coins <=68

Demonstration

Coin values 1,1,1,2,2,2,5,5,5,10,10,10,20,20,20,50,50,50



Control Abstraction

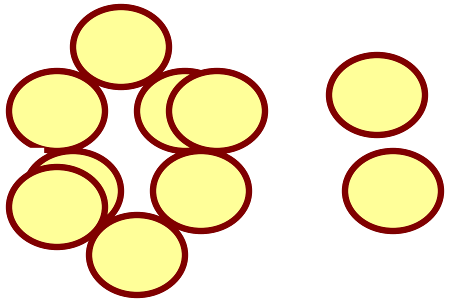
```
Algorithm Greedy(a,n)
{solution :=Ø
for i:=1 to n do
{x:=select(a)
if(feasible(solution,x) then
solution:=union(solution,x);}
return solution;}
```

Different Paradigms

- Subset Paradigm
- Ordering Paradigm

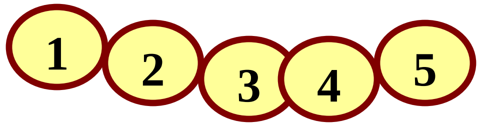
Subset Paradigm

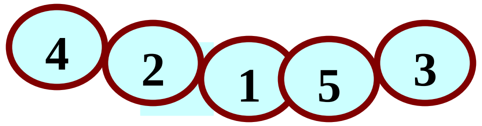
- Determine a subset of the given *n* inputs



Ordering Paradigm

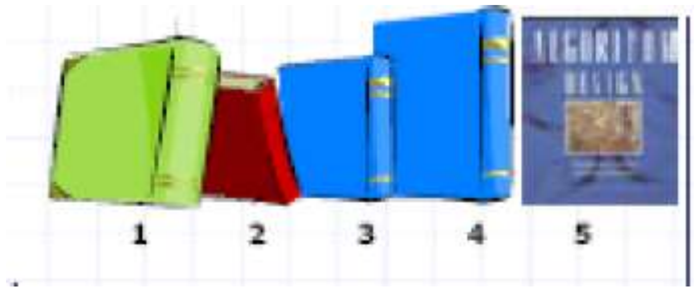
- Determine an ordering of all the inputs





Knapsack Problem

There are n objects to be placed in a knapsack of capacity M. Each object i contributes a profit Pi and weight Wi. The total profit $\sum_{1 \le i \le n} P_i X_i$ should be maximized subject to the constraint that total weight $\sum_{1 \le i \le n} W_i X_i \le M$ where M is the capacity of the Knapsack . Pi And Wi are positive numbers where $0 \le X_i \le 1$ and $1 \le i \le n$



Knapsack Problem

general knapsack problem with n=3 m=20

Pi = { 25,24,15} Wi={18,15,10} for i= 1 ,2,3

I	1	2	3	
Pi	25	24	15	
Wi	18	15	10	
PiXi	25	2/15*24=3.2	0	tot=28.2
PiXi	0	10/15*24=16	15	tot=31
$\frac{Pi}{Wi}$	$\frac{25}{18}$	$\frac{24}{15}$	$\frac{15}{10}$	
	=1.11	=1.6	=1.5	
PiXi	0	24	5/10*15=7.5	tot=31.5

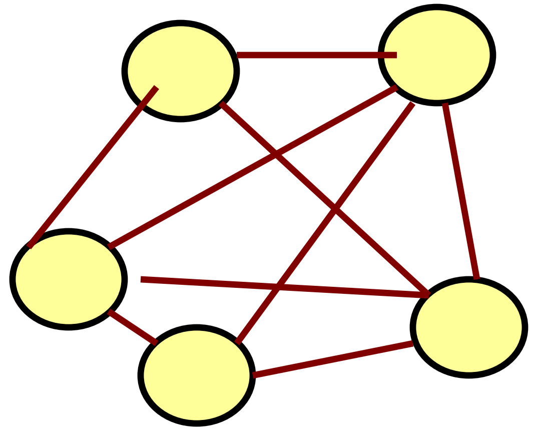
Solution **Xi** = **{0 , 1 , 0.5}**

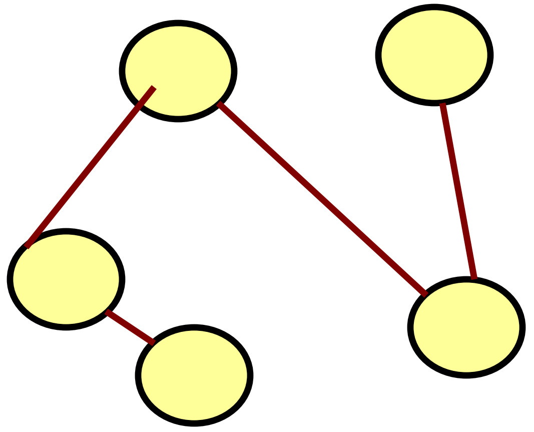
Algorithm

```
Algorithm greedy knapsack(M,n,P,W)
{for i =1 to n do
x(i) := 0; Re := M;
Full := false;  i := 1;
While(i<=n and not(full))
{   if(Wi<=Re) then
    {Xi=1; Re:=Re-Wi; i++; }
    else (full=true)}
if(i<=n) then Xi=Re/Wi;} }
```

Spanning Tree

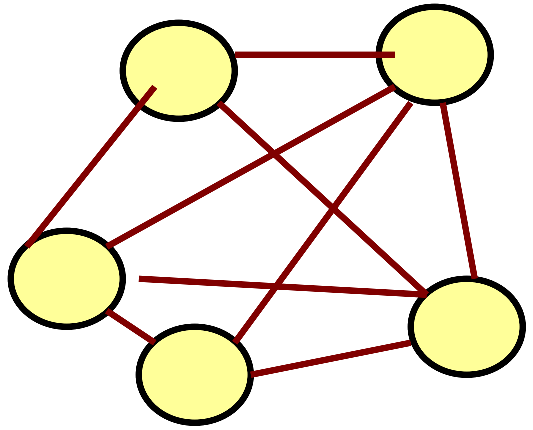
- Let G=(V,E) be an undirected connected graph.
- A sub-graph T=(V,E’) of G is a spanning tree of G if T is a tree

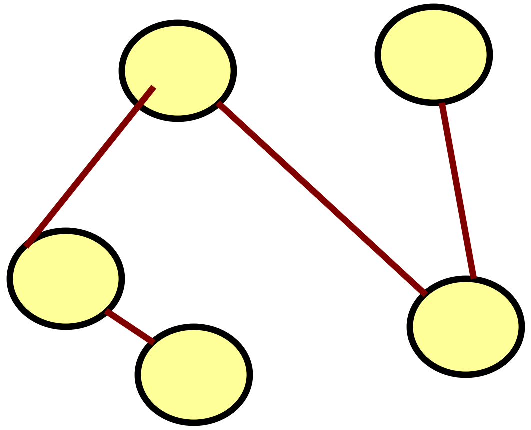




Conditions - Spanning Tree

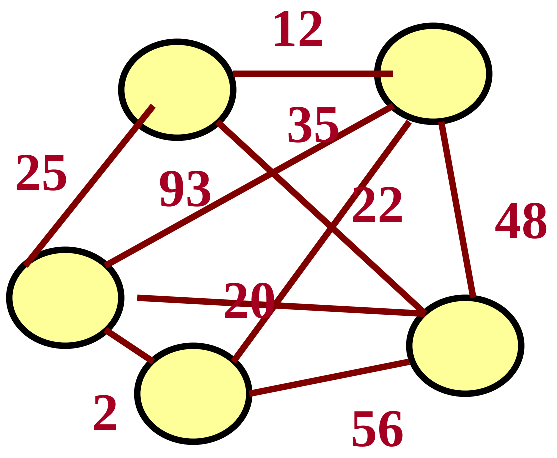
- For T a subgraph of G to be a spanning tree.
  - T should contain all vertices V in G
  - T should be a tree- there are no cycles



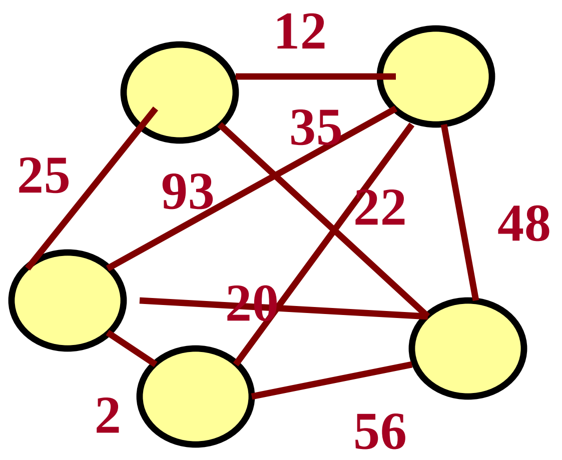


Minimum Cost Spanning Tree

- A Spanning tree which has minimum cost
  - – Sum of edge costs



PRIMS Algorithm



- Constraint
  - Each vertex once
  - Tree

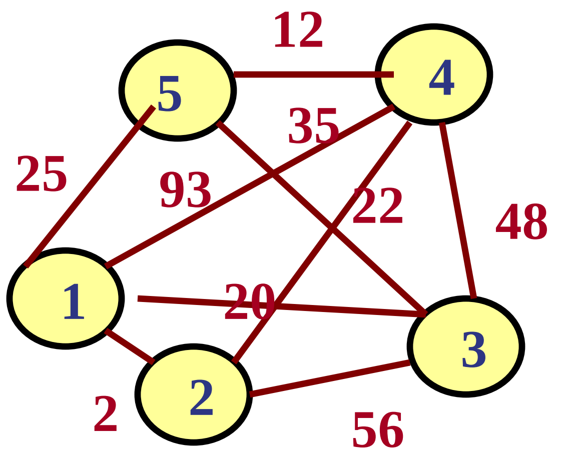
- Selection Principle
  - Vertex not present in Tree
  - Vertex which can be connected with lowest cost to the spanning tree is next vertex selected

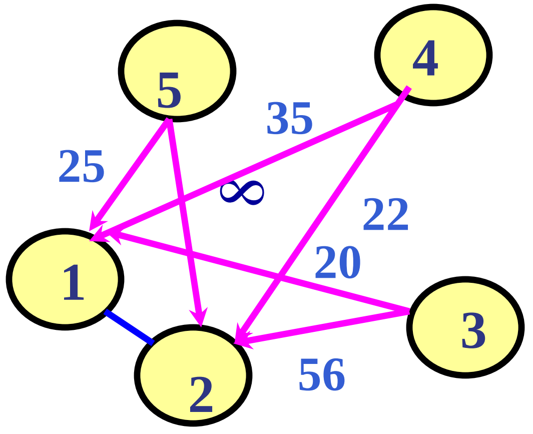
Informal Algorithm

```
ST:=∅
Y:= {u,v}, where <u,v> is the edge with lowest cost
ST:={(u,v)}
While the instance is not solved do
{Select a vertex in V-Y that is nearest to Y
Add the corresponding edge to ST
If Y= V then mark instance ‘solved’ // included all
}                                        the vertices
```

Near Array

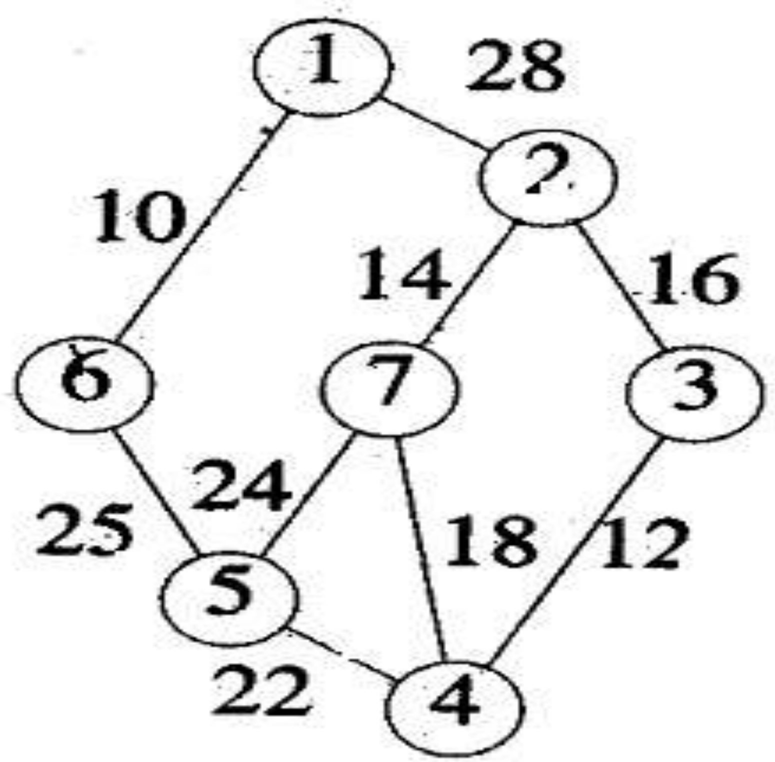
- For every un included a vertex j which is not in the spanning Tree, near[j] represents a vertex which is nearest to vertex j in the spanning tree.
- Edge(j,near[j]) represents the shortest path of connecting j to the current spanning tree

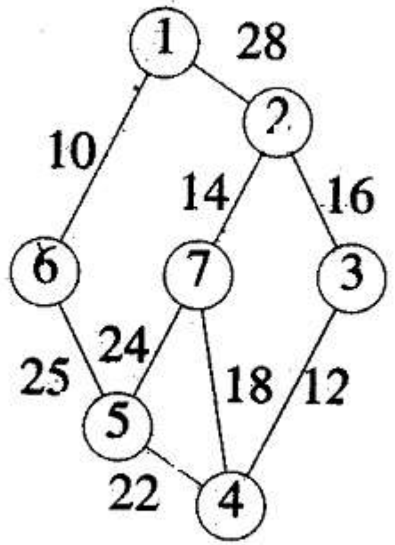




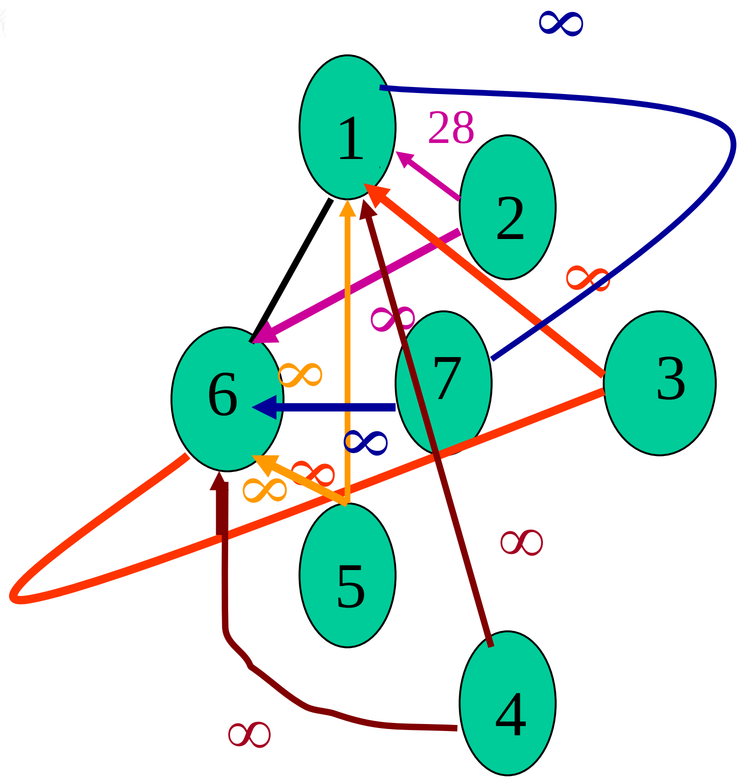
3	1	20
4	2	22
5	1	25

Example

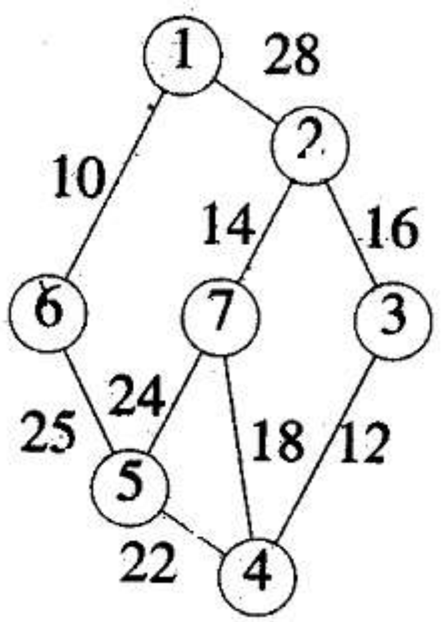




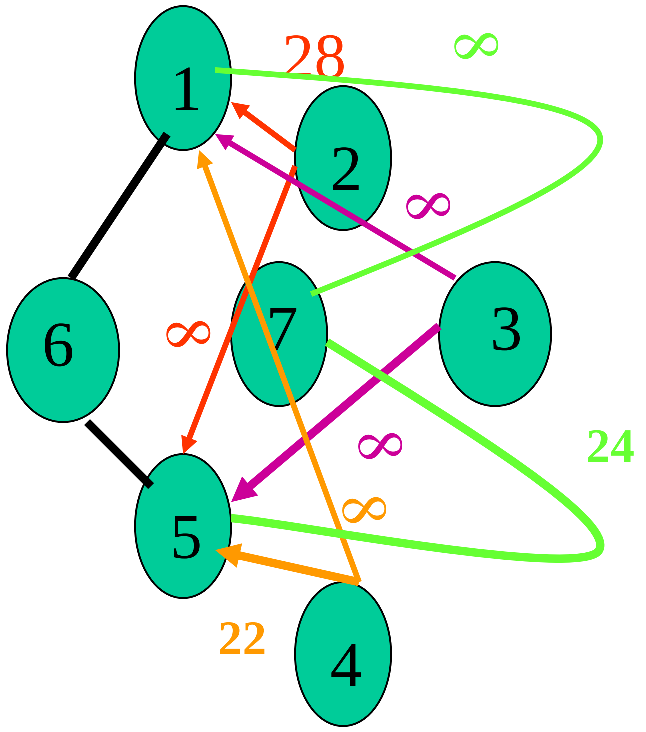
Demonstration



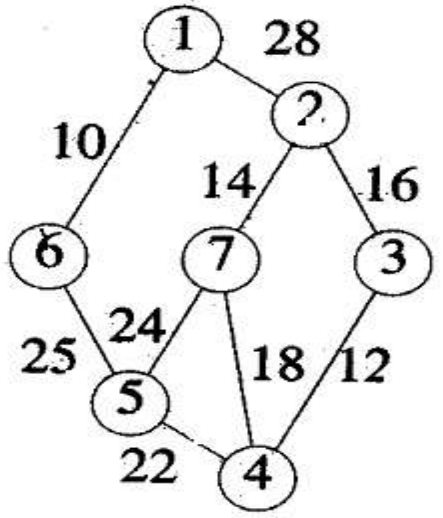
1	-	-
2	1	28
3	∞	∞
4	∞	∞
5		25
6	6	25 -
7		
	∞ Near	∞ Cost



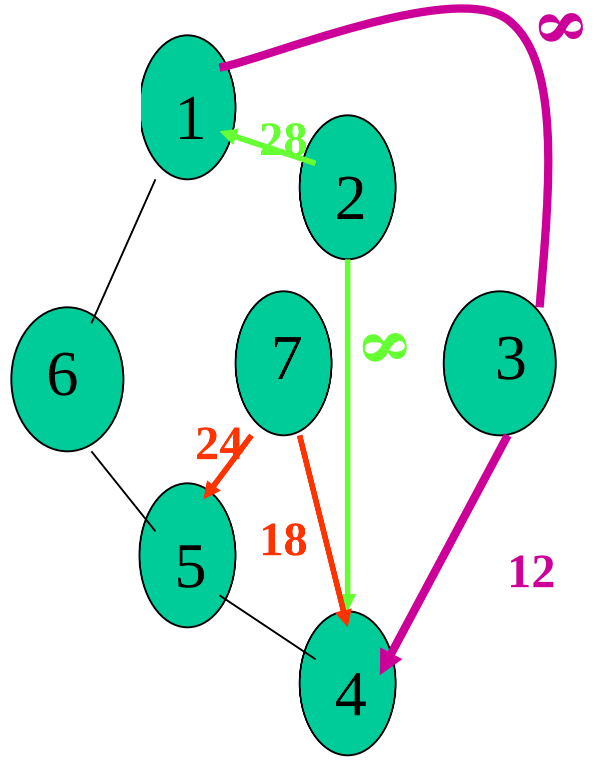
Demonstration



1	-	-
2	1	28
3	∞	∞
4	5	22
5	5	2
6	-	-
7	5	24
	Near	Cost

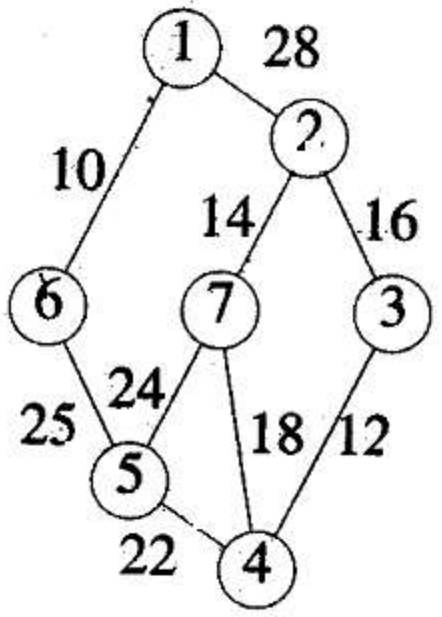


Demonstration

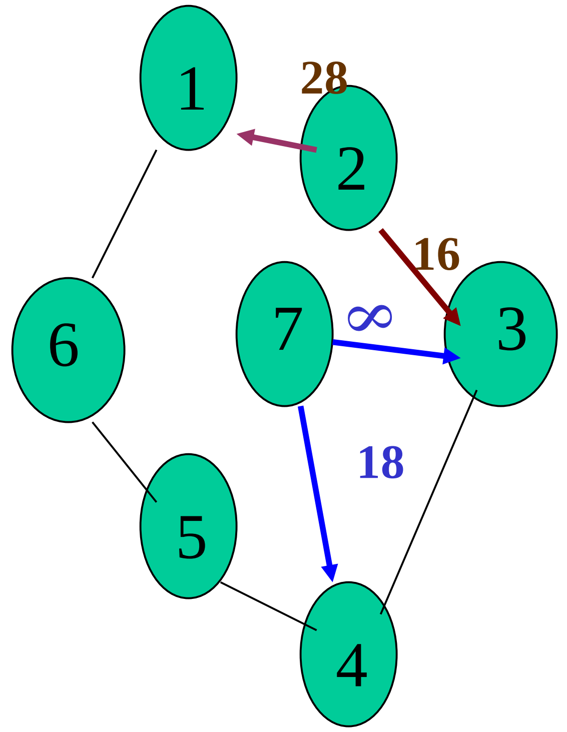


1	-	-
2	1	28
3	4	12
4	-	-
5	-	-
6	-	-
7	4	18

Near cost

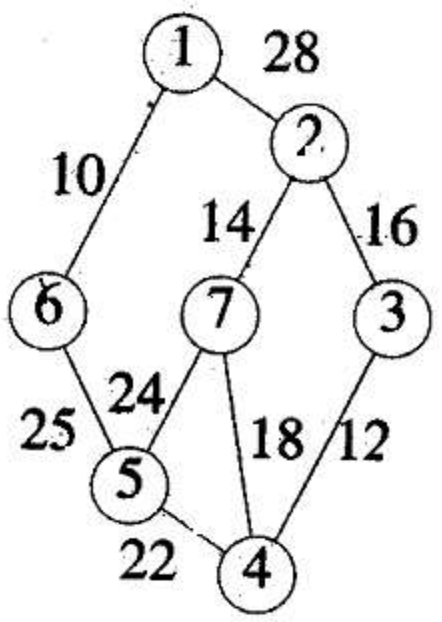


Demonstration

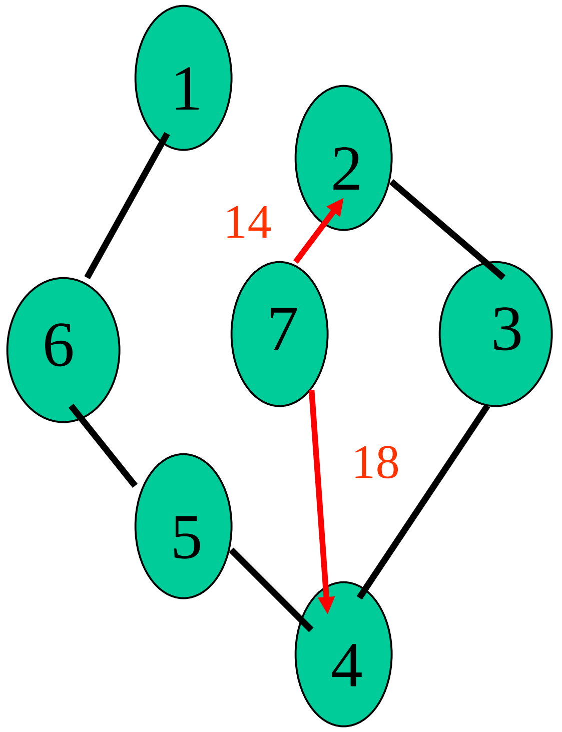


1	-	-
2	3	16
3	-	-
4	-	-
5	-	-
6	-	-
7	4	18

Near cost

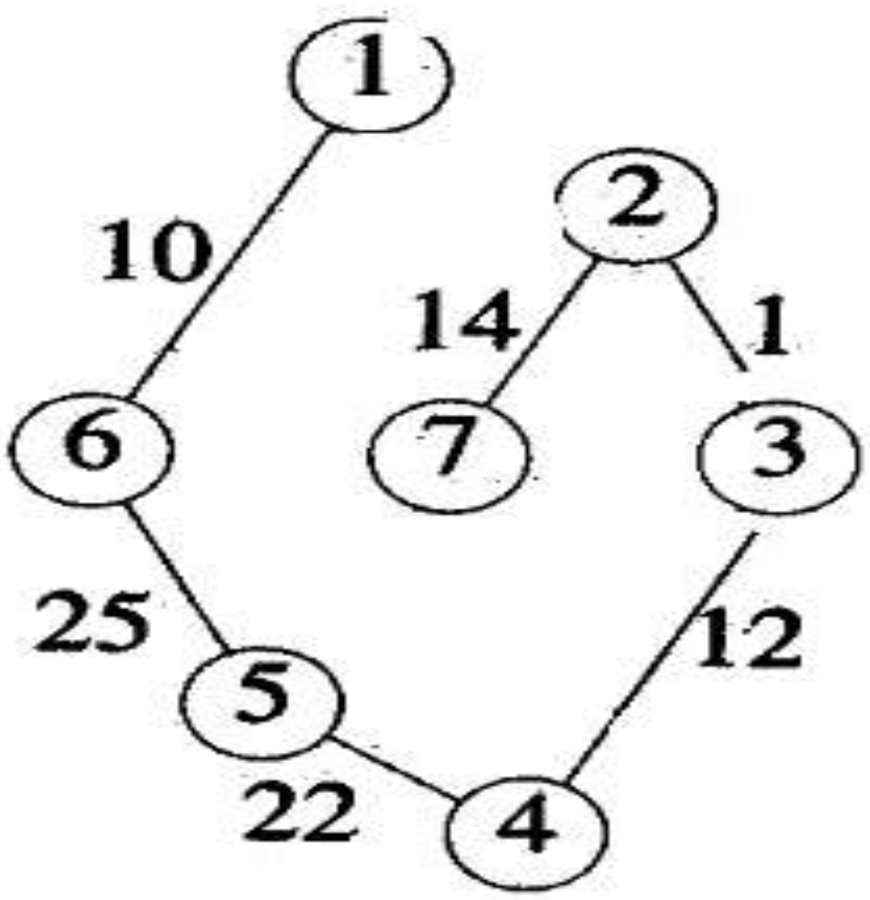


Demonstration



1	-	-
2	-	-
3	-	-
4	-	-
5	-	-
6	-	-
7	2	14
	Near	cost

Solution



ALGORITHM ANALYSIS AND DESIGN

BACKTRACKING

A Short List Of Categories

- Algorithm types we will consider include:
  - Simple recursive algorithms
  - Divide and conquer algorithms
  - Greedy algorithms
  - Dynamic programming algorithms
  - Backtracking algorithms
  - Branch and bound algorithms

Introduction

- Backtracking is used to solve problems in which a sequence of objects is chosen from a specified set so that the sequence satisfy some criterion.

Backtracking- When

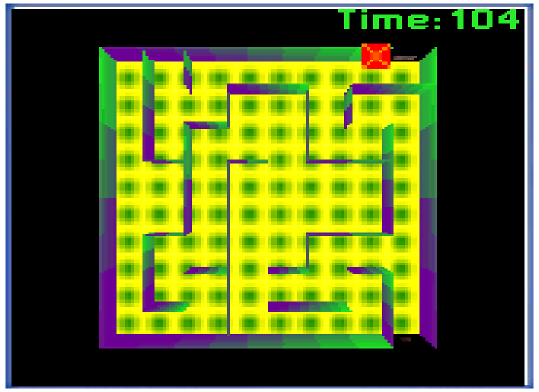
- There is a sequence of decisions to be made, from a number of available choices, where
  - suffient information is not available on the best choice
  - Each decision leads to a new set of choices
  - Some sequence of choices (possibly more than one) may be a solution to your problem

Backtracking- How

- Backtracking is a systematic method of trying out various sequences of decisions, until you find one that “works”

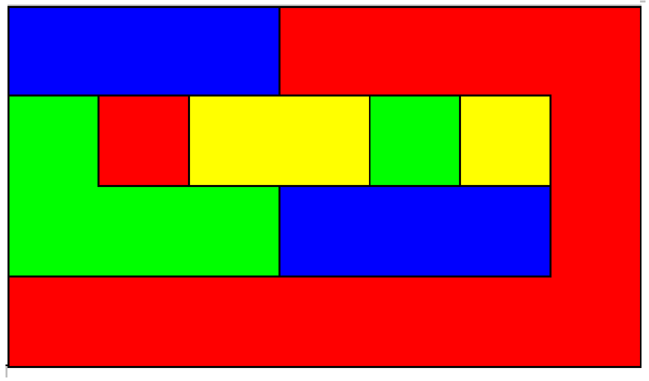
Solving A Maze

- Given a maze, find a path from start to finish
- At each intersection, you have to decide between three or fewer choices:
  - Go straight
  - Go left
  - Go right



- Sufficient information not available on the best choice
- Each choice leads to another set of choices
- One or more sequences of choices may (or may not) lead to a solution

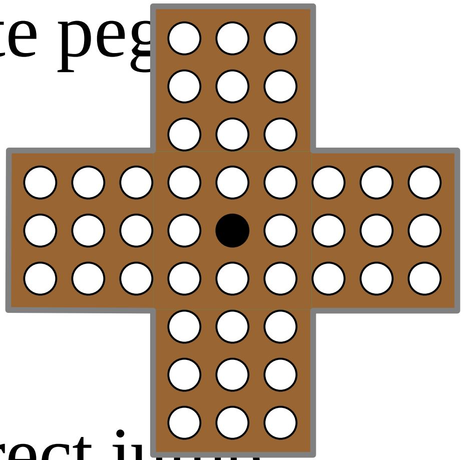
Coloring A Map



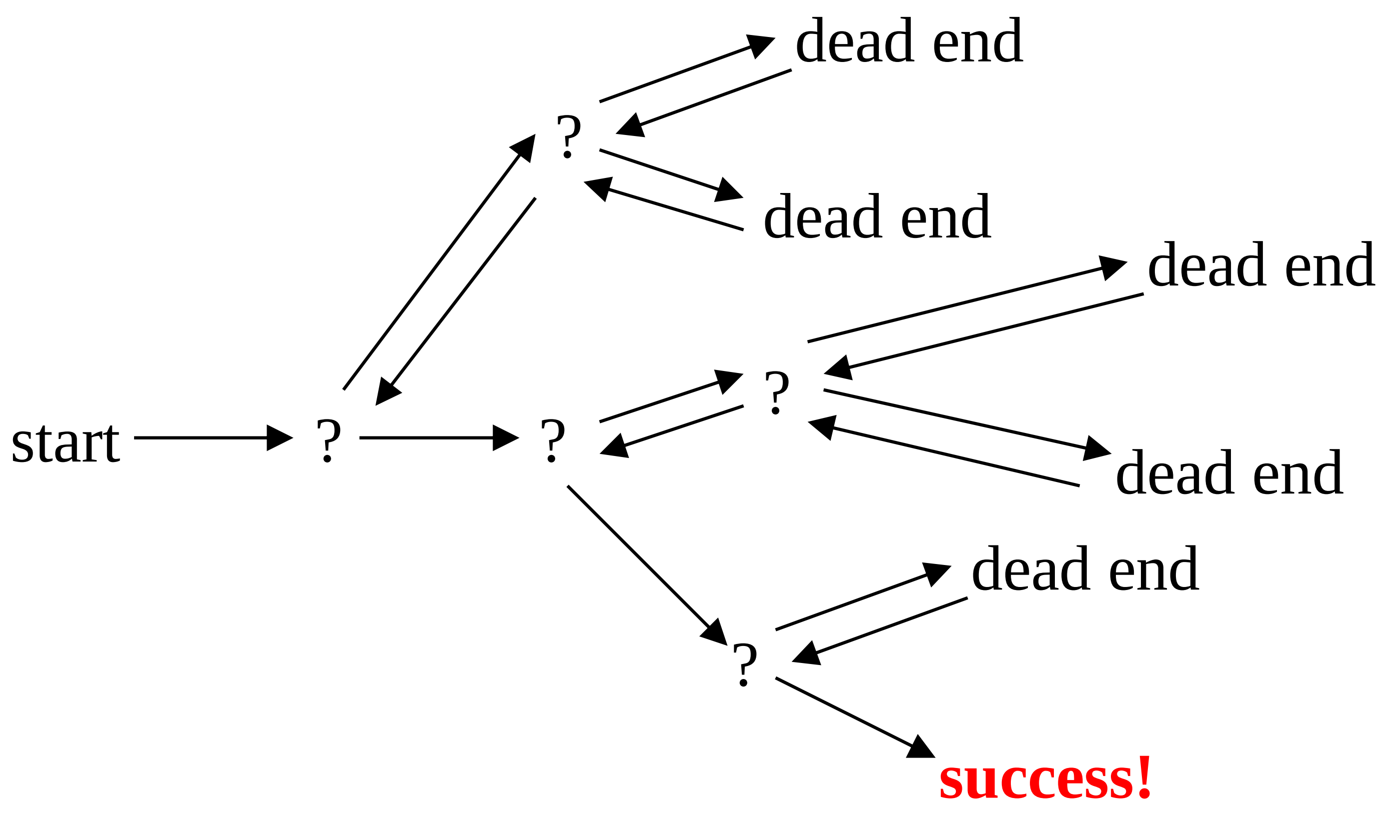
- You wish to color a map with not more than four colors
  - red, yellow, green, blue
- Adjacent countries must be in different colors
- You don’t have enough information to choose colors
- Each choice leads to another set of choices
- One or more sequences of choices may (or may not) lead to a solution

Solving A Puzzle

- All holes except the middle filled with white pegs
- We can jump over one peg with another
- Jumped pegs are removed
- The aim is to remove all but the last peg
- Sufficient information not available on correct jump
- Each choice leads to another set of choices
- One or more sequences of choices may (or may not) lead to a solution

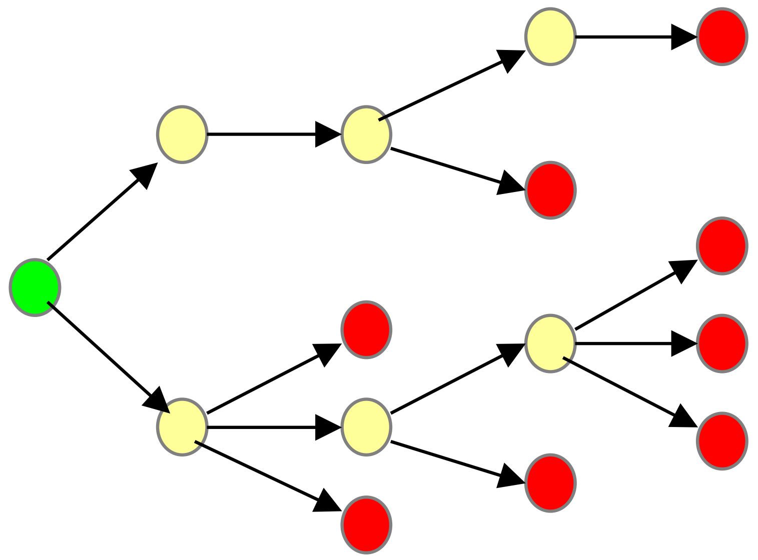


Backtracking- Demonstration



Representation

The decision sequences can be represented by a tree



Three kinds of nodes:

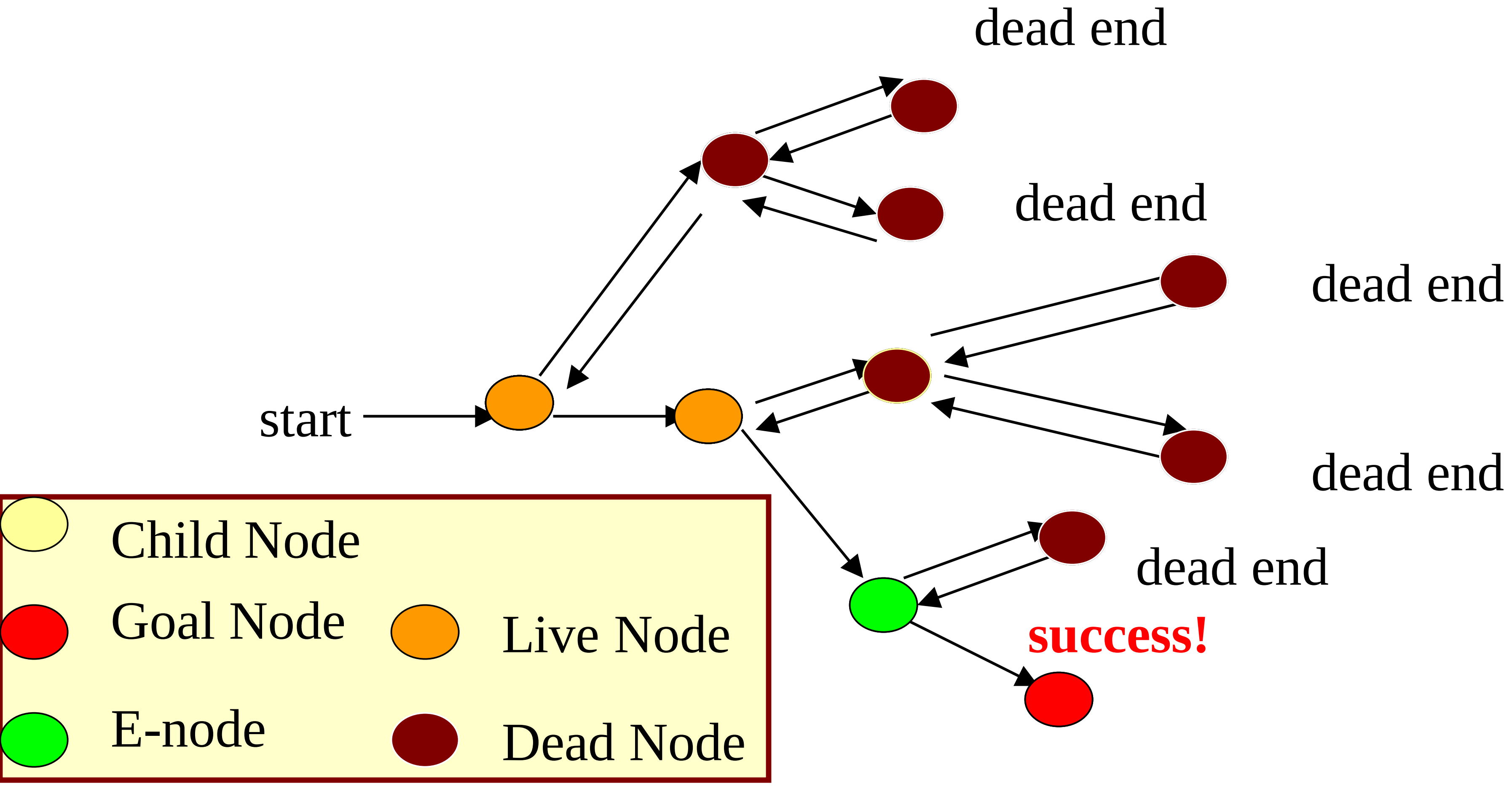
- 🟢 The (one) root node
- 🟡 Internal nodes
- 🔴 Leaf nodes

Backtracking can be thought of as searching a tree for a particular "goal" leaf node

Types Of Nodes

- Live node
  - A node which has been generated, and all of its children have not yet been generated
- Dead node
  - A generated node which is not to be expanded further or all of whose children have been generated.
- E-node
  - The live node whose children are being generated.

Types Of Nodes - Demonstration



Constraints

- Explicit constraints
  - Rules that restrict each element to be chosen from the given set
  - Only tuples that satisfy explicit constraints will appear in the tree
- Implicit constraints
  - Rules that specify how each elements in a tuple should be related
  - Tuples in the tree that satisfy the implicit constraints represent a goal state

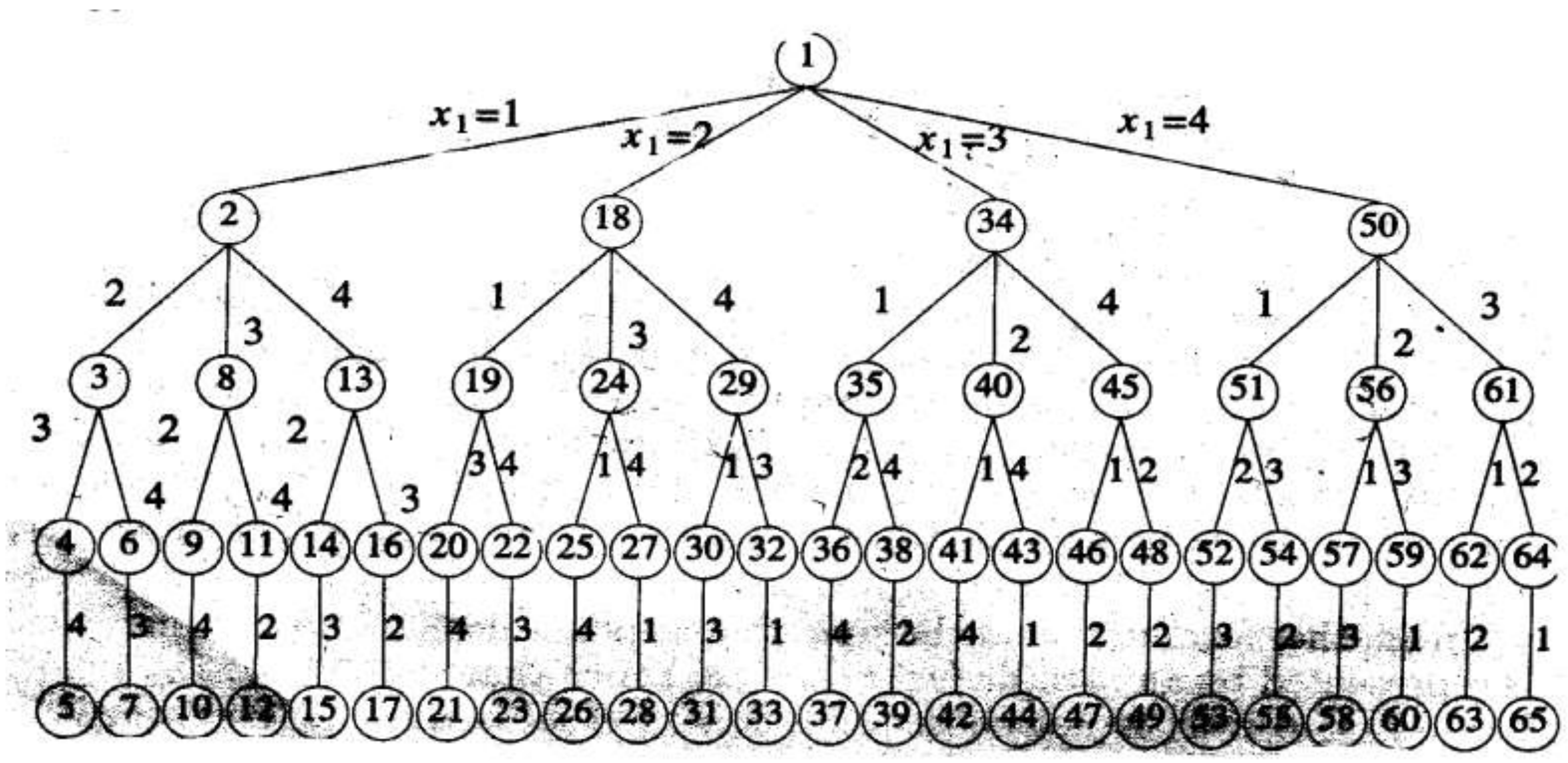
Constraints-Example

- Let x_i represents i th decision
- Explicit constraints
  - Restrict the possible values of xi
  - Eg:- $x_i = 0$ or 1; $x_i \in \{li..ui\}$
- Implicit constraints
  - Specify how the different xi's are related
  - Eg:- $x_i < x_{i+1}$; $N = \sum_{k=1}^{i-1} X_k$

Solution Space

- Set of all tuples that satisfy the explicit constraints
- Represented using a permutation tree- state space tree
- The edges labeled by possible values of xi
  - Edges from level 1 to level 2 nodes specify the values for x1
- Defined by all paths from the root to a leaf node

State Space Tree



Solution Space Contd..

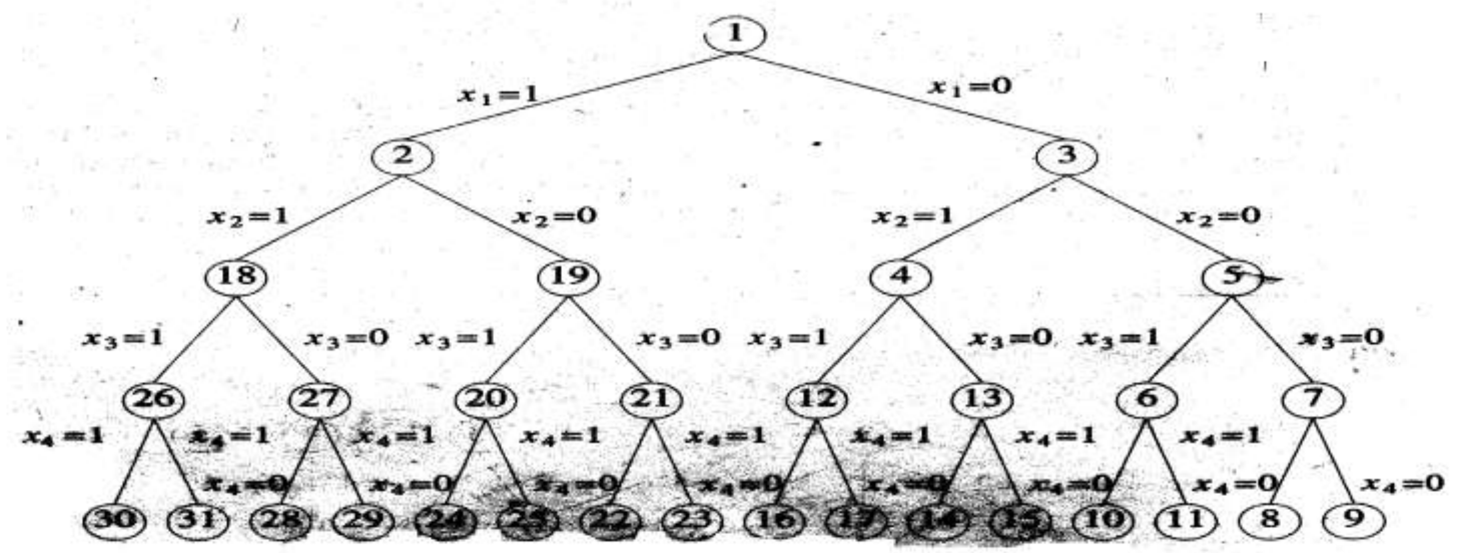
- Each level in the tree defines a problem state
- Paths from root to other nodes are known as state spaces
- Solution states- problem states s for which the path from the root to s defines a tuple in the solution space
- Answer states - solution states s for which the path from the root defines a solution from the set of solutions

State Space Tree -Organization

- A problem can be specified in 2 ways
- Fixed tuple formulation
  - The state space organization is called static trees
- Variable tuple formulation
  - State space tree of this type is known as dynamic tree

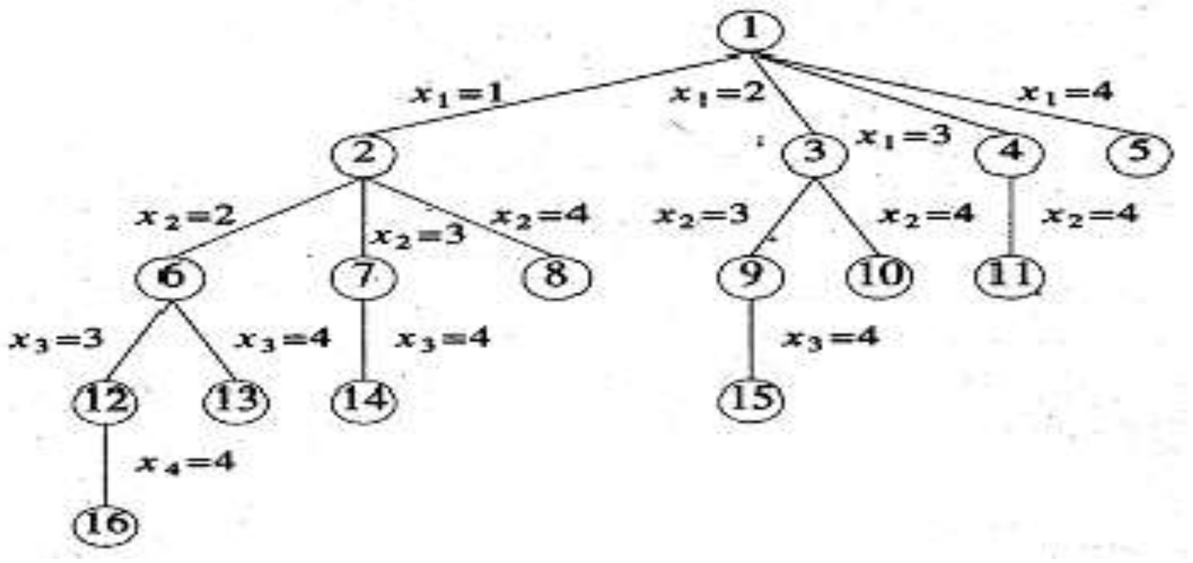
Fixed Tuple Formulation

- Edges from level i nodes to level i+1 nodes are labeled with the value of xi, which is either 0 or 1
- All paths from root to leaf node define the solution space



Variable Tuple Size Formulation

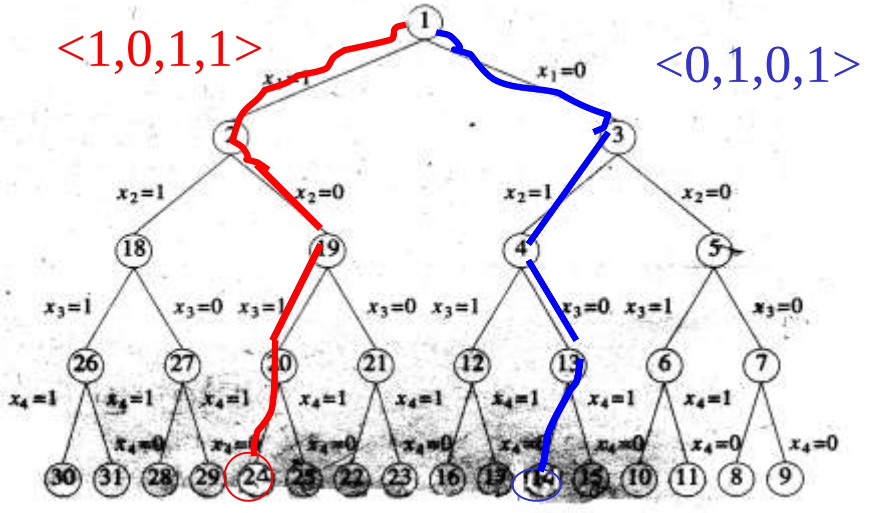
- Size of solution tuples is not fixed.
- Each tuple will contain a subset of the xi’s in the order in which they are included instead of specifying xi=0 or 1.



Comparison

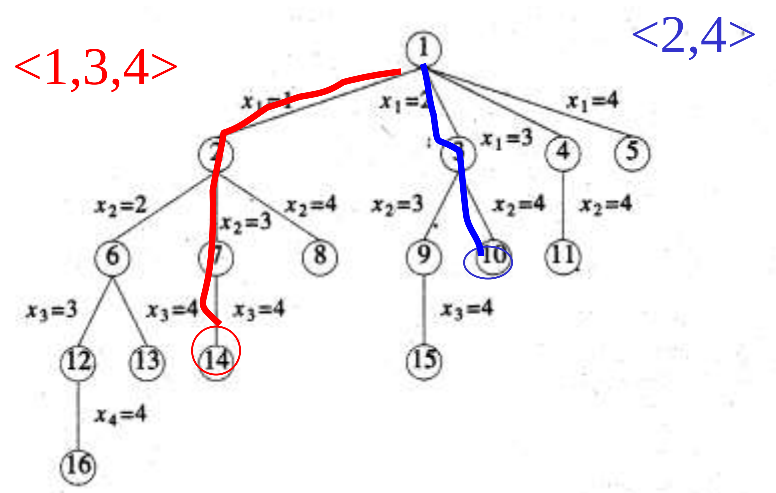
Fixed tuple formulation

- Solution size is fixed
- Only the leaf nodes can be solution states



Variable tuple formulation

- Solution size is variable
- Any intermediate node can be the solution state



Solving The Problem

- Fix a state space tree organization
- Systematically generate the problem states starting from the root
- Verify each problem state whether it is a solution state or an answer state

State Space Tree Generation

- Backtracking uses depth first generation of the state space tree with some bounding function
- Bounding functions are used to kill live nodes without generating all their children
- Killing should be done after ensuring that all the required answer states are generated

The Backtracking Algorithm

Backtracking algorithm is quite simple
it requires "exploring" each node, as follows:

- To "explore" node N:

```
1. If N is a goal node, return "success"
2. If N is a leaf node, return "failure"
3. For each unbounded child C of N,
    3.1. Explore C
        3.1.1. If C is successful, return "success"
4. Return "failure"
```

Backtracking -Impementation

- Generating Function
  - Generate next child, given the parent
- Bounding function
  - Check whether the particular path can lead to an answer state and returns false If it cannot.

Control Abstraction

```
Algorithm backtrack(n)
{k=1;
while (k#0) do
{
if (there remains an untried x[k] ε T(x[1],x[2],. . . ,x[k-1]) and
    $B_k$(x[1],x[2], . . .,x[k]) is true) then
{
if x[1]. …x[k] is a path to an answer node) then write x[1:k]);
k:=k+1;}
else k:=k-1;}}
```

Recursive Formulation

```
Algorithm Backtrack(k)
  //This schema describes the backtracking process using
  //recursion. On entering, the first k-1 values
  //x[1],x[2],….,x[k-1] of the solution vector
  //x[1 : n] have been assigned. x[] and n are global.
  {     for (each x[k] ε T(x[1],……,x[k-1]) do
        {if($B_k$ (x[1],x[2],……x[k] $\neq$ 0 ) then
   {if(x[1],x[2],….,x[k]  is a path to an answer node)
    then write (x[1 : k]);
  if(k < n) then Backtrack(k + 1);}}}
```

Efficiency Of Backtracking

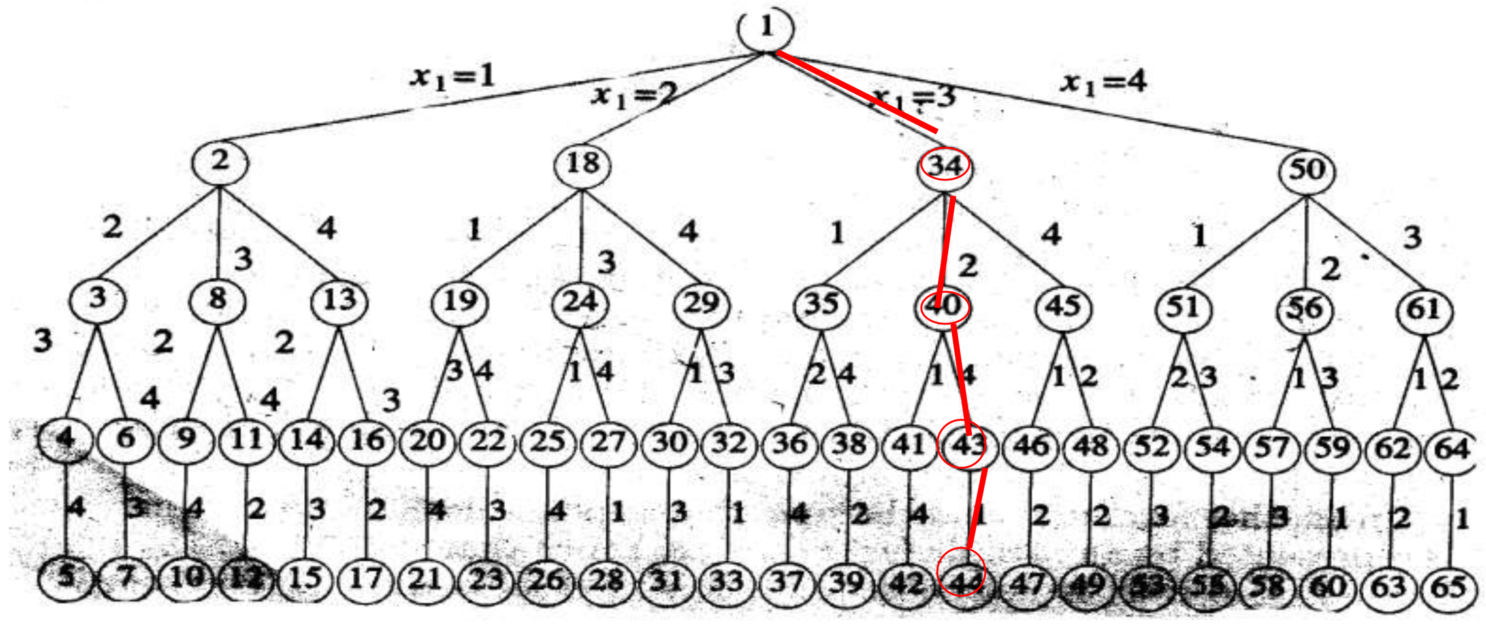
- Time taken to generate next x_k
- Number of x_k satisfying the explicit constraints
- Time required for computing the bounding function
  - A bounding function is said to be good if it reduces the number of nodes drastically.
  - A good bounding function may take much computing time
- Number of x_k satisfying b_k
  - Trade off between the number of nodes removed and computing time requirement.
  - The ultimate aim is to reduce the total time requirement

Analysis

- For some problem instances, if the order of generation of children is changed, the number of nodes to be generated before finding the solution can be drastically reduced.
- Depending on the problem instance, the backtracking algorithm will generate O(n) nodes to n! nodes.
- The importance of backtracking lies in its ability to solve some instances with large n in a very small amount of time.

Estimate Of Nodes Generated

The idea is to generate a random path in the state space tree
Monte carlo method



Random Path Generation

- Let x be a node at level i in the random path
- Generate children of x
- Apply bounding functions to each child.
- Let mi be number of unbounded children of x.
  - unbounded children at level 1 is m1, level 2 m2 ...
- Find an estimate of the number of unbounded nodes at level i+1

Estimate of Unbounded Nodes

- Estimate of the number of unbounded nodes at level i+1
  - estimate of unbounded nodes at level 1 = m1
  - estimate of unbounded nodes at level 2 = m1*m2
  - Estimate of unbounded nodes at level i+1 = m1*m2*- - -*mi

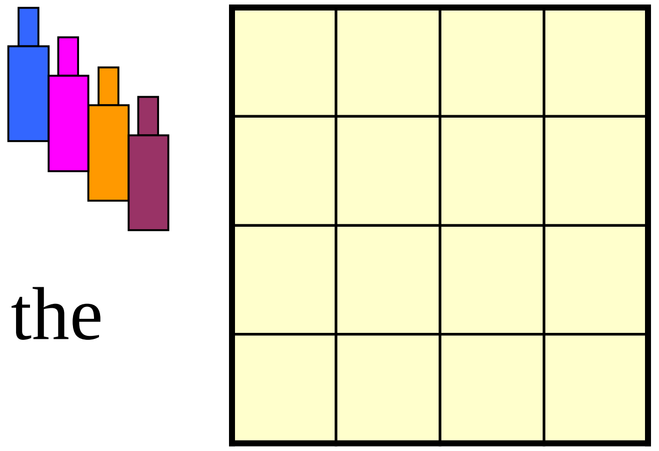
Monte Carlo Method

- Random path
  - starts from the root of the tree and
  - is continued till
    - a leaf node is reached
    - or a node with all bounded children is reached
- Each time a child node of the current node is selected as the random node at the next level.
- The total nodes considering all levels is equal to $1+m_1+m_1m_2+- - -+m_1*m_2*--m_n$

Algorithm

```
Algorithm Estimate
{ // r is the children at a particular level; m –total nodes;k -level
k:=1;m:=1;r:=1;
repeat
{Tk= {x[k] / x[k] ε T(x[1],……,x[k-1] and Bk(x[1],. . ,x[k]) is true
    };
If (size(Tk)=0 ) then return m;
r:= r* size(tk); m:=m+r;
X[k]:= choose(Tk); k:=k+1;
}until (false);}
```

4 Queens Problem

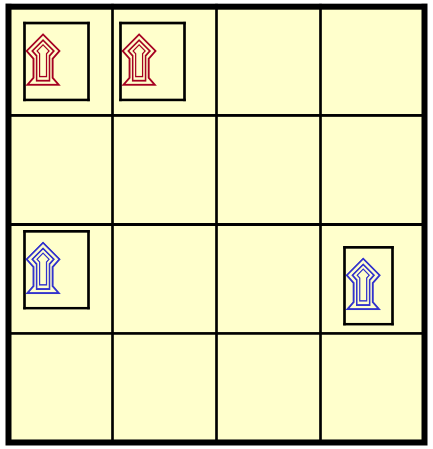


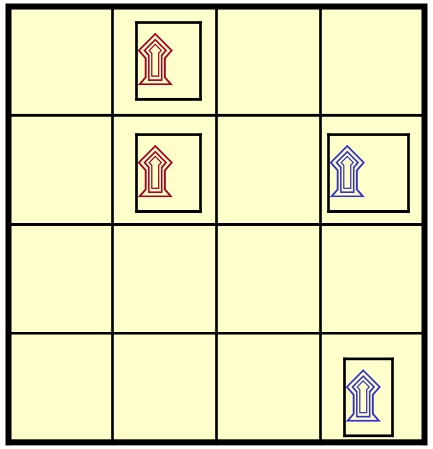
- There are 4 queens that are to be placed on a 4x4 chessboard
- Queens should be placed on the chessboard such that no two queens can attack each other.
- Two queens attack each other when they are in the
  - Same row
  - Same column or
  - Same diagonal.

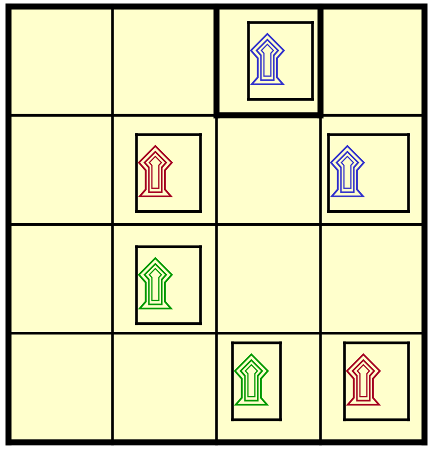
Attacking Positions

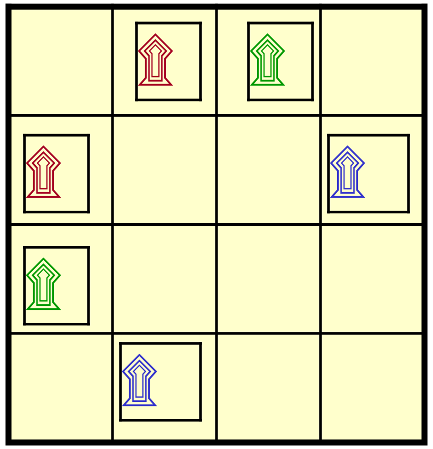
Two queens attack each other when they are in the

- Same row
- Same column or
- Same diagonal.


Rule 1

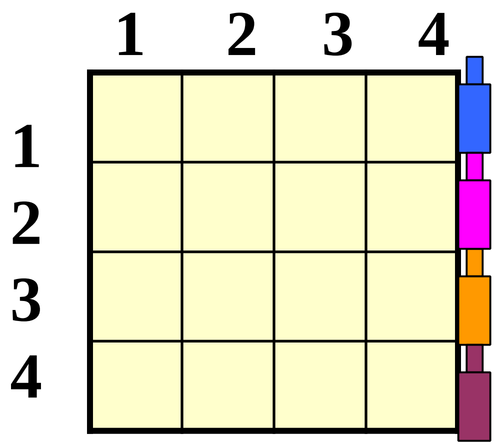

Rule 2


Rule 3.a.


Rule 3.b.

Problem Specification

- The queens are numbered from 1 to 4.
- Rows and columns in the chess board are also numbered from 1 to 4.
- Each row can contain exactly one queen.
- So it is assumed that queen i is placed in row i.



Queen 1 is placed in row 1

Queen 2 is placed in row 2

Queen 3 is placed in row 3

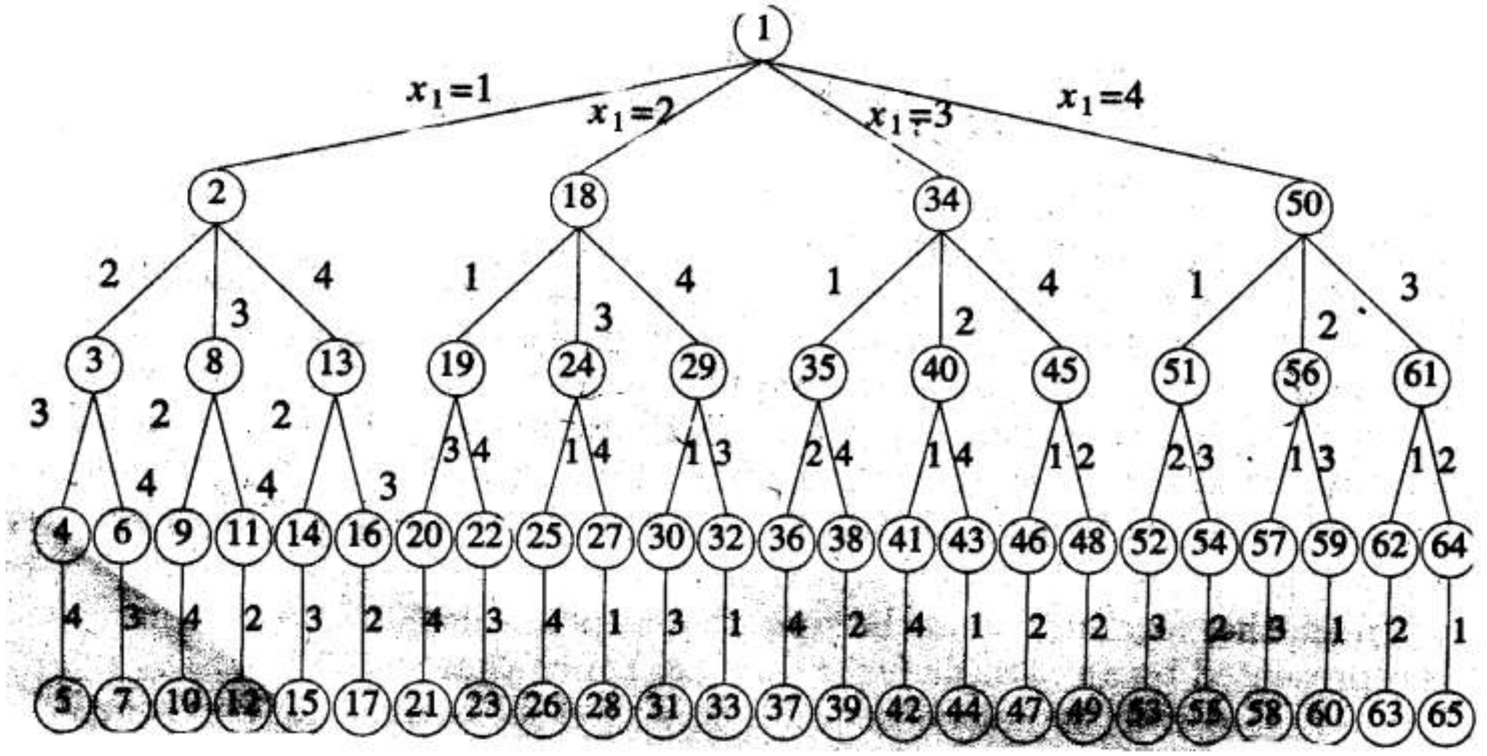
Queen 4 is placed in row 4

Solution Space

- The solution space consists of 4^4 4 tuples.
- The implicit constraint
  - no two queens are in the same column
    - No two xi’s can be the same
  - no two queens can be on the same diagonal.
- Column constraint restricts the solution space to consist of combinations of 4 tuples.
- So the solution space gets reduced to 4! Tuples.
- Second constraint specify that no two xi’s should be on the same diagonal.

Solution Space

- Number of tuples generated after considering column constraint - 4! Tuples



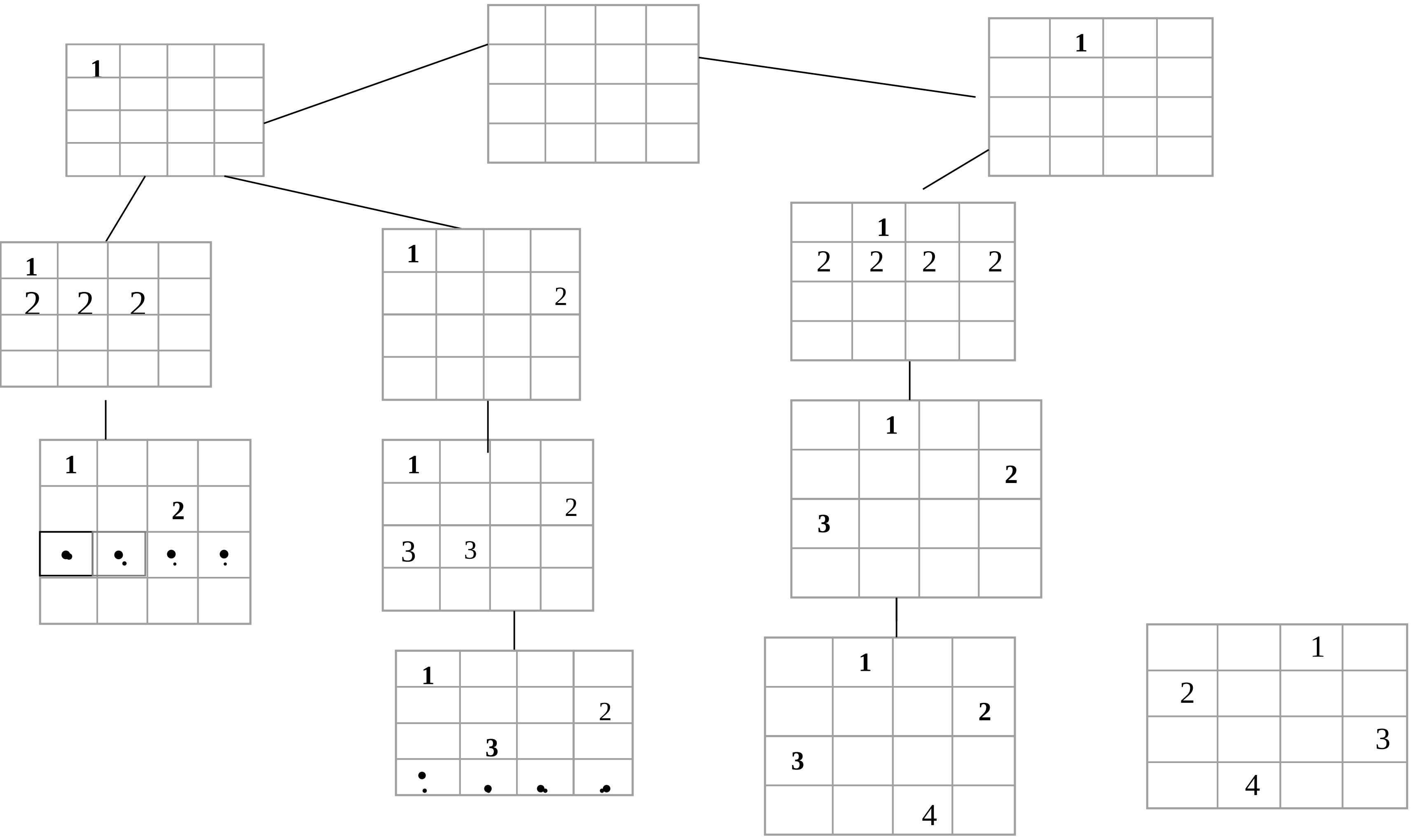
Problem State-representation

1			
			2
·	3		

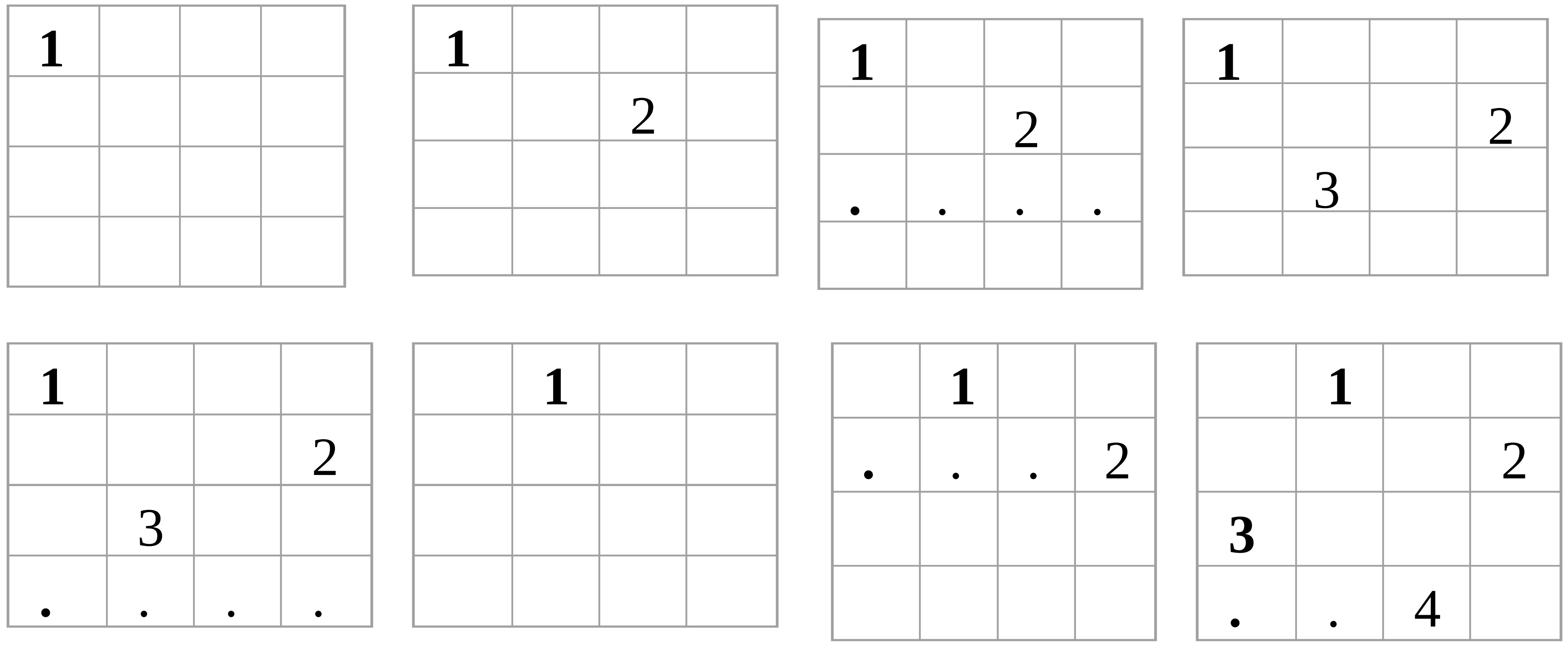
Queen i is represented by the number i

· represents an unsuccessful attempt

4 Queens Problem- Tree Formation



4 Queens - Formation Of Solution



N Queens Problem

- Queens problem can be generalized to n queens problem.
  - N queens
  - N X N chess board
  - Solution a sequence of n decisions

Bounding Function

- The bounding function should return a value true if the current queen can be placed at a particular position.

- i^{th} queen can be placed at position k if none of the previous queens are in the same column or in the same diagonal.

- Same column verification by checking whether xi=k for all previous queens.

Same Diagonal Verification

- QUEENS at (i,j) and (k,l)
- same forward diagonal if i-j=k-l or j-l = i-k (1)
- same backward diagonal if i+j=k+l or j-l = k-i (2)
- combining 1&2 we can write

  |j - l |= |i – k|
- so two queens lies on the same diagonal if and only if

  |j - l |= |i – k|

Formation Of Bounding Function

- same column verification by checking whether xi=k for all previous queens.

- so two queens lies on the same diagonal if and only if

  $|j - l| = |i - k|$

Each queen j,x[j] with queen(i,k)

if ((x[j]=i) or (abs(x[j]-i)= abs(j-k)))

Bounding Function -Place

Each queen j,x[j] with queen i,k
if ((x[j]=i) or (abs(x[j]-i)= abs(j-k)))

```
Algorithm place(k,i)
{
for j:=1 to k-1 do
if ((x[j]=i) or (abs(x[j]-i)= abs(j-k) ))
then return false;
return true;
}
```

Algorithm

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

Complexity

- The number of 8 tuples generated considering placement of queens in distinct rows and columns is 8!.
- Estimated number of nodes in the state space tree

  1+8+8*7+8*7*6+ - - =

$$1 + \sum_{j=0}^{7} \sum_{i=0}^{j} (8 - i) = 69,281$$

Complexity Contd..

- In practice the number of nodes generated will be very very less than this.
- From Experiments it is seen that it will be approximately 3% on the average.

Sum Of Subsets

- Given n positive numbers/weights
- Find the combinations of numbers whose sums are m.

Sum Of Subsets - Example

n=6 {w1,w2,……….w6}

= {5,10,12,13,15,18}

m = 30

Solution {5,10,15} {5,12,13} {12,18}

Sum Of Subsets- Formulation

Numbers are arranged and considered in the non decreasing order.

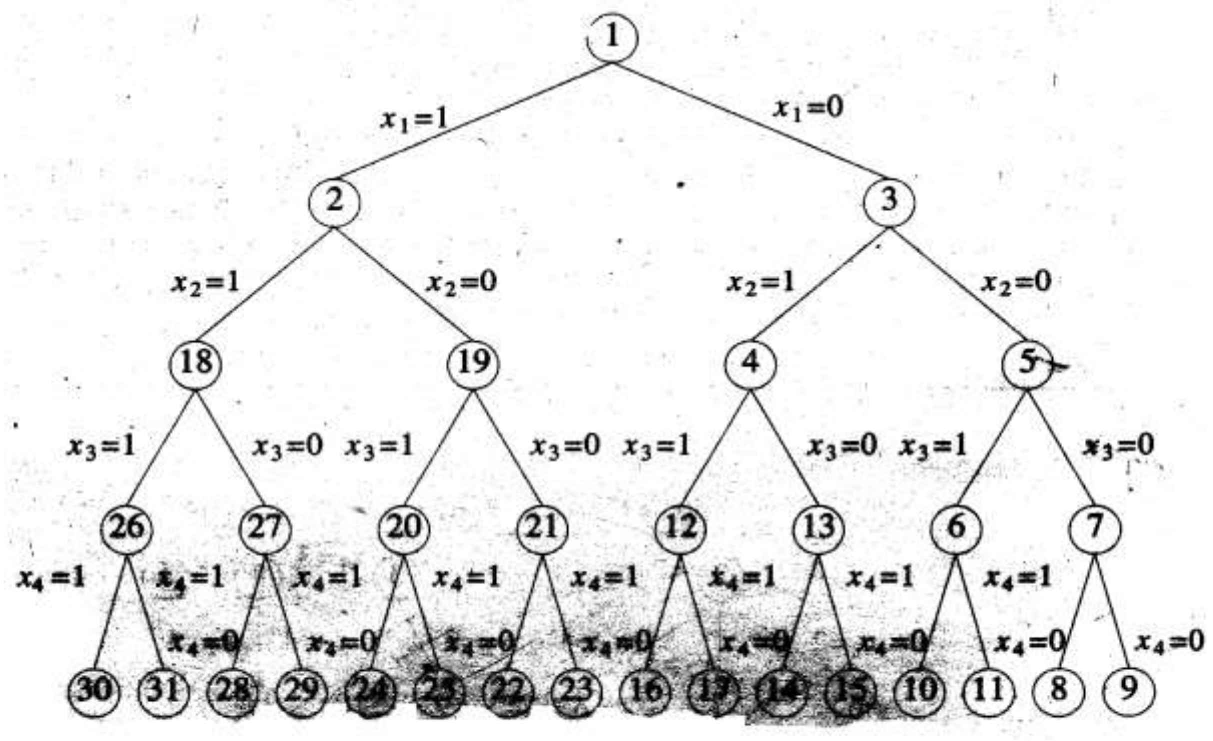
It can be formulated in 2 ways.

In fixed tuple each xi=0 or 1

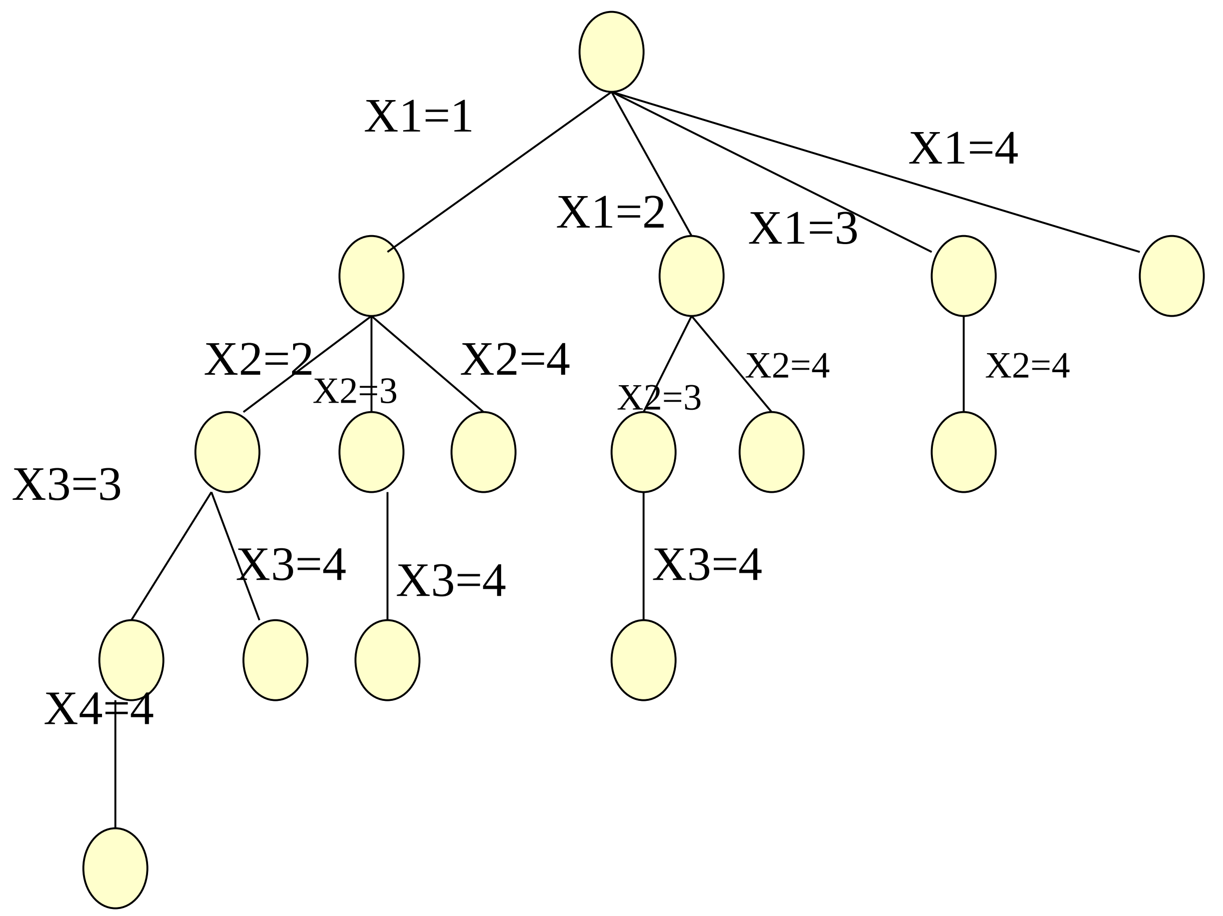
In variable tuple xi=1..n depending on the next number selected.

In fixed tuple formulation the state space tree

State Space Tree
Fixed Tuple Formulation



State Space Tree
Variable Tuple Formulation



Formation Of Bounding Function

Example: {w1,w2,……….w6} = {5,10,12,13,15,18} & m= 30

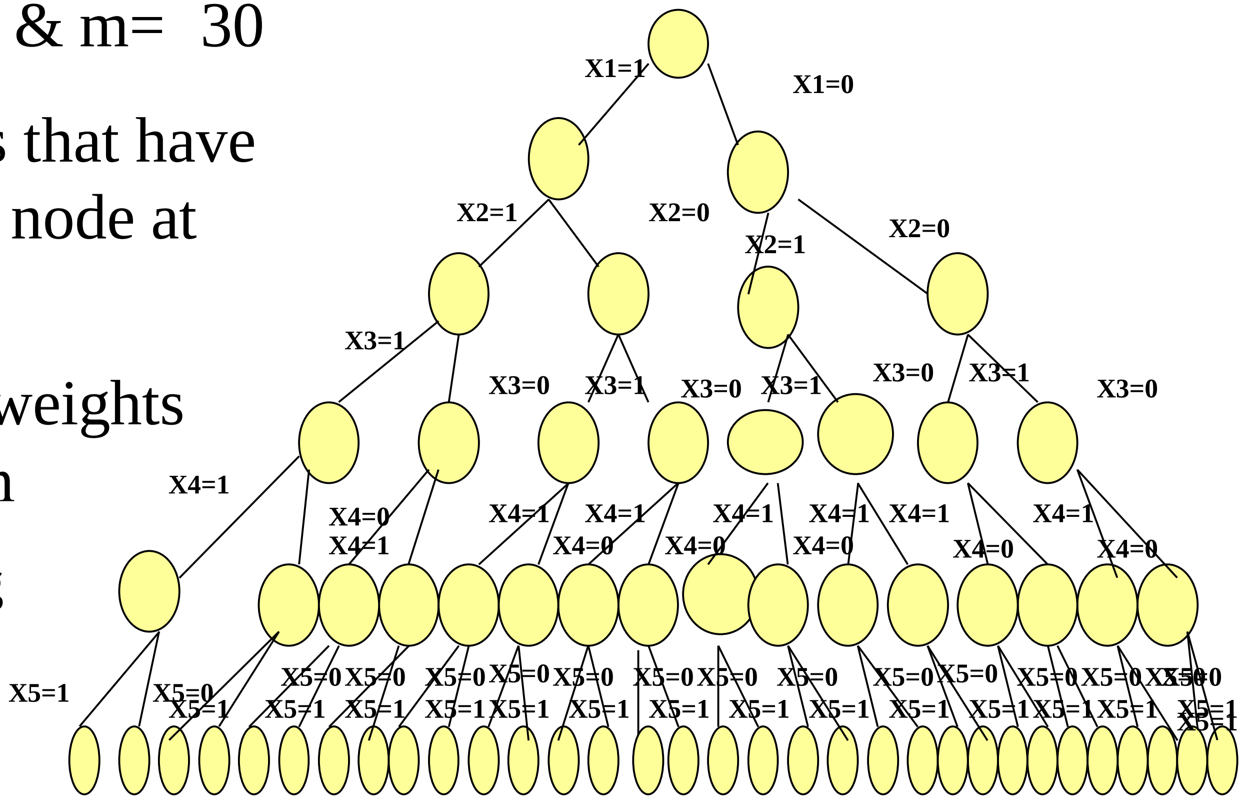
s : the sum of weights that have been included up to a node at level i.

r : sum of remaining weights wi+1 ,wi+2………wn

The node is nonpromising

if s+w $_{i+1}$ > m

if s+r < m



Bounding Function

- A node can lead to an answer state if

$$\sum_{i=1}^{k} WiXi + \sum_{i=k+1}^{n} Wi >= m$$

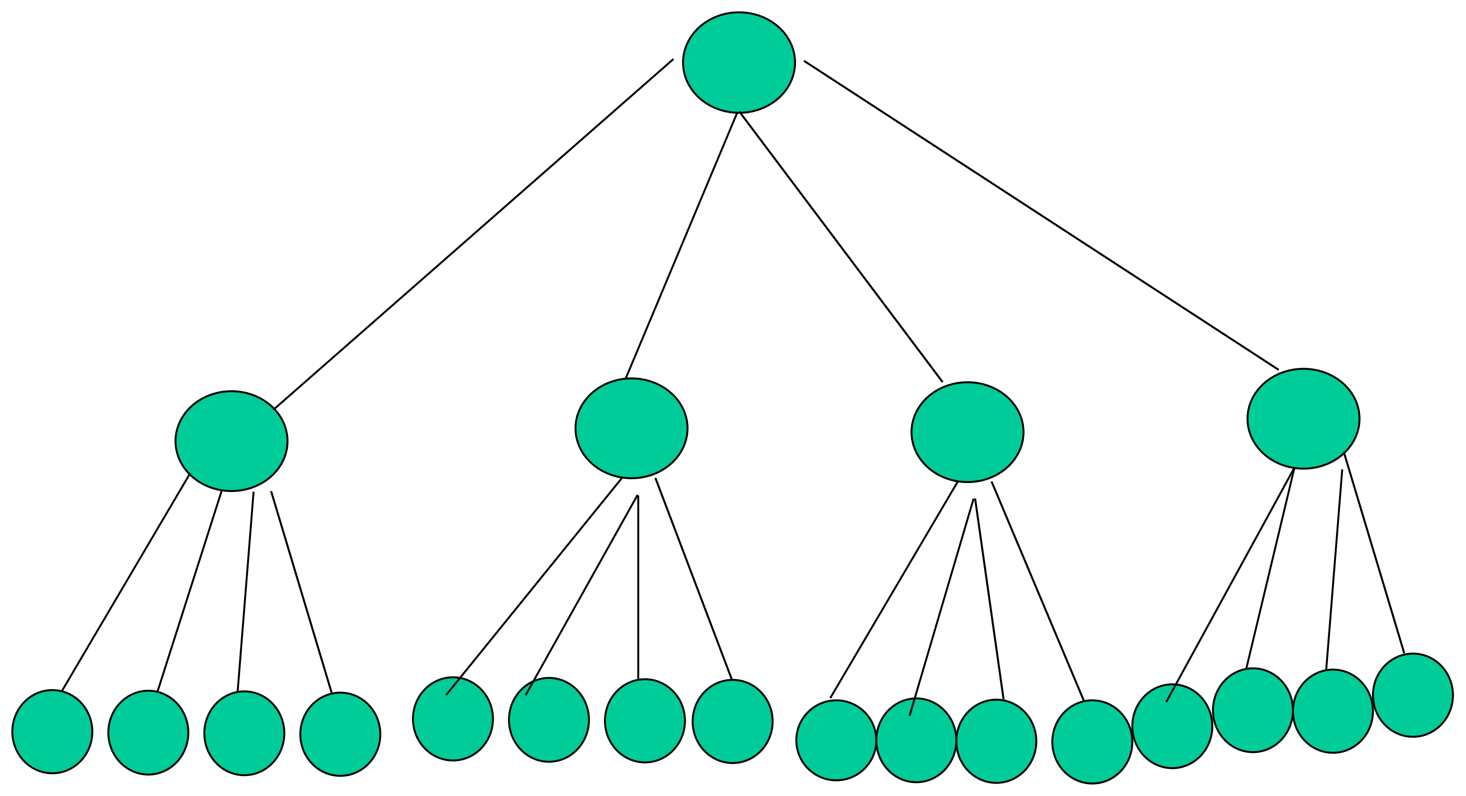
$$\sum_{i=1}^{k} WiXi + W_{k+1} >= m$$

- Bk is true iff

- Computation of the sums $\sum_{i=1}^{k} WiXi$ and $\sum_{i=k+1}^{n} Wi$ each time when it is required, can be avoided by maintaining two sums s for current aggregate and r for remaining aggregate.

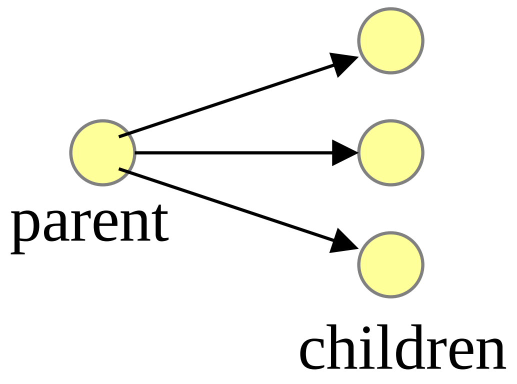
Algorithm

```
Algorithm sumofsub(s,k,r)
{//generate left child with xk=1 fixed tuple
   formulation
x[k]:=1;
if (s+w[k]=m) then write (x[1:k]);
else if (s+w[k]+w[k+1]<=m)
then sumofsub(s+w[k],k+1,r-w[k]);
if ((s+r-w[k]>=m) and (s+w[k+1]<=m)) then
{x[k]:=0;
sumofsub(s,k+1,r-w[k]);}}
```

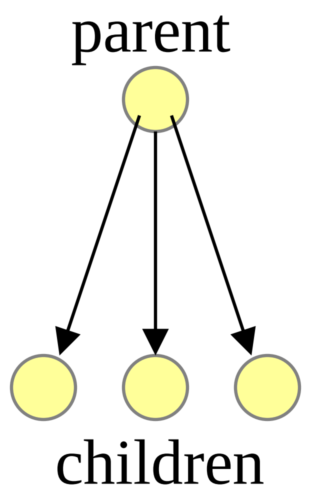


Terminology II

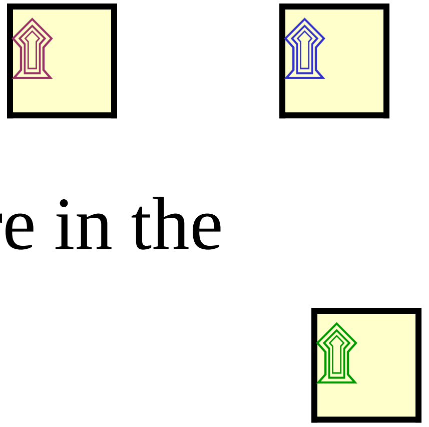
- Each non-leaf node in a tree is a parent of one or more other nodes (its children)
- Each node in the tree, other than the root, has exactly one parent



Usually, however, we draw our trees *downward*, with the root at the top

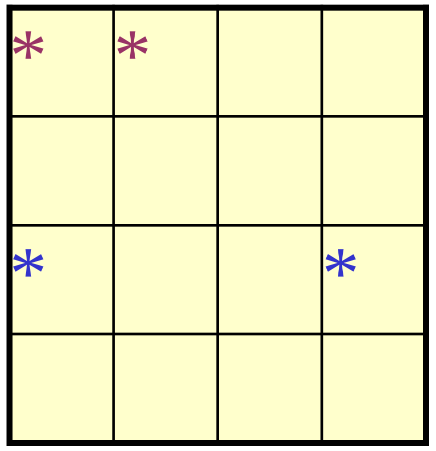


ATTACKING POSITIONS

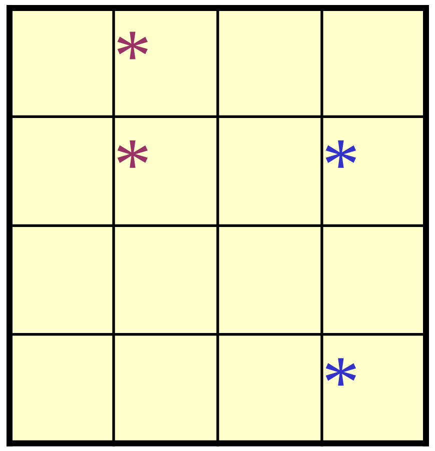


Two queens attack each other when they are in the

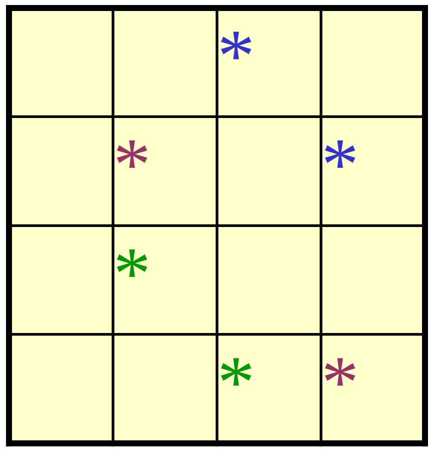
- Same row
- Same column or
- Same diagonal.



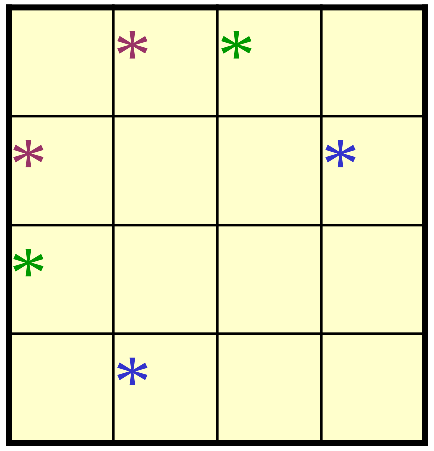
Rule 1



Rule 2



Rule 3.a.



Rule 3.b.