Introduction to Data Structures and Algorithms

Pham Quang Dung and Do Phan Thuan

Computer Science Department, SoICT,
Hanoi University of Science and Technology.

July 8, 2016

First example

Find the longest subsequence of a given sequence of numbers

- Given a sequence $s = \langle a_1, \ldots, a_n \rangle$
- a subsequence is $s(i,j) = \langle a_i, \ldots, a_j \rangle$, $1 \leq i \leq j \leq n$
- weight $w(s(i,j)) =$
$$\sum_{k=i}^{j} a_k$$
- Problem: find the subsequence having largest weight

Example

- sequence: -2, 11, -4, 13, -5, 2
- The largest weight subsequence is 11, -4, 13 having weight 20

Direct algorithm

- Scan all possible subsequences $\binom{n}{2} = \frac{n^2+n}{2}$
- Compute and keep the largest weight subsequence

```java
public long algo1(int[] a){
  int n = a.length;
  long max = a[0];
  for(int i = 0; i < n; i++){
    for(int j = i; j < n; j++){
      int s = 0;
      for(int k = i; k <= j; k++)
        s = s + a[k];
      max = max < s ? s : max;
    }
  }
  return max;
}
```

Direct algorithm

Faster algorithm

- Observation: $\sum_{k=i}^{j} a[k] = a[j] + \sum_{k=i}^{j-1} a[k]$

```
public long algo2(int[] a){
  int n = a.length;
  long max = a[0];
  for(int i = 0; i < n; i++){
    int s = 0;
    for(int j = i; j < n; j++){
      s = s + a[j];
      max = max < s ? s : max;
    }
  }
  return max;
}
```

Recursive algorithm

- Divide the sequence into 2 subsequences at the middle $s = s_1 :: s_2$
- The largest subsequence might
  - be in s_1 or
  - be in s_2 or
  - start at some position of s_1 and end at some position of s_2
- Java code:

```java
private long maxSeq(int i, int j){
  if(i == j) return a[i];
  int m = (i+j)/2;
  long ml = maxSeq(i,m);
  long mr = maxSeq(m+1,j);
  long maxL = maxLeft(i,m);
  long maxR = maxRight(m+1,j);
  long maxLR = maxL + maxR;
  long max = ml > mr ? ml : mr;
  max = max > maxLR ? max : maxLR;
  return max;
}
public long algo3(int[] a){
  int n = a.length;
  return maxSeq(0,n-1);
}
```

Recursive algorithm

```
private long maxLeft(int i, int j){
  long maxL = a[j];
  int s = 0;
  for(int k = j; k >= i; k--){
    s += a[k];
    maxL = maxL > s ? maxL : s;
  }
  return maxL;
}
private long maxRight(int i, int j){
  long maxR = a[i];
  int s = 0;
  for(int k = i; k <= j; k++){
    s += a[k];
    maxR = maxR > s ? maxR : s;
  }
  return maxR;
}
```

Dynamic programming

General principle

- Division: divide the initial problem into smaller similar problems (subproblems)
- Storing solutions to subproblems: store the solution to subproblems into memory
- Aggregation: establish the solution to the initial problem by aggregating solutions to subproblems stored in the memory

Dynamic programming

Largest subsequence

- Division:
  - Let s_i be the weight of the largest subsequence of $a_1, \ldots, a_i$ ending at a_i
- Aggregation:
  - $s_1 = a_1$
  - $s_i = \max\{s_{i-1} + a_i, a_i\}, \forall i = 2, \ldots, n$
  - Solution to the original problem is $\max\{s_1, \ldots, s_n\}$
- Number of basic operations is n (best algorithm)

Dynamic programming

```
public long algo4(int[] a){
  int n = a.length;
  long max = a[0];
  int[] s = new int[n];
  s[0] = a[0];
  max = s[0];
  for(int i = 1; i < n; i++){
    if(s[i-1] > 0) s[i] = s[i-1] + a[i];
    else s[i] = a[i];
    max = max > s[i] ? max : s[i];
  }
  return max;
}
```

Analyzing algorithms

- Resources (memory, bandwithd, CPU, etc.) required by the algorithms
- Most concern is the computational time
- Input size: number of items in the input
- Running time: measured in term of the number of primitive operations performed

Analyzing algorithms

- algo1: $T(n) = \frac{n^3}{6} + \frac{n^2}{2} + \frac{n}{3}$
- algo2: $T(n) = \frac{n^2}{2} + \frac{n}{2}$
- algo3:
  - Count the number of addition ("+") operation $T(n)$

$$T(n) = \begin{cases} 0 & \text{if} \quad n = 1 \\ T(\frac{n}{2}) + T(\frac{n}{2}) + n & \text{if} \quad n > 1 \end{cases}$$

  - By induction: $T(n) = n \; log_2 n$
- algo4: $T(n) = n$

Analyzing algorithms

- Worst-case running time: the longest running time for any input of size n
- Best-case running time: the shortest running time for any input of size n
- Average-case running time: probabilistic analysis (make assumption of a distribution of the input) yields expected running time

Order of growth

- Consider only leading term of the function
- Ignore constant coefficient
- Example
  - $an^3 + bn^2 + cn + d = \Theta(n^3)$

Asymptotic notations

- Given a fucntion $g(n)$, we denote:
  - $\Theta(g(n)) = \{f(n) : \exists c_1, c_2, n_0 \text{ s.t. } 0 \leq c_1 g(n) \leq f(n) \leq c_2 g(n), \forall n \geq n_0\}$
  - $\mathcal{O}(g(n)) = \{f(n) : \exists c, n_0 > 0 \text{ s.t. } f(n) \leq cg(n), \forall n \geq n_0\}$
  - $\Omega(g(n)) = \{f(n) : \exists c, n_0 > 0 \text{ s.t. } cg(n) \leq f(n), \forall n \geq n_0\}$
- Examples
  - $10n^2 - 3n = \Theta(n^2)$
  - $10n^2 - 3n = \mathcal{O}(n^3)$
  - $10n^2 - 3n = \Omega(n)$

Experiments studies

- Write a program implementing the algorithm
- Execute the program on a machine with different input sizes
- Measure the actual execution times
- Plot the results

Analysis of algorithms

Shortcomings of experiments studies

- Need to implement the algorithm, sometime difficult
- Results may not indicate the running time of other input not experimented
- To compare two algorithms, it is required to use the same hardware and software environments.

Analysis of algorithms

Asymptotic algorithm analysis

- Use high-level description of the algorithm (pseudo code)
- Determine the running time of an algorithm as a function of the input size
- Express this function with asymptotic notations

Analysis of algorithms

- Sequential structure: P and Q are two segments of the algorithm (the sequence $P; Q$)
  - $\text{Time}(P; Q) = \text{Time}(P) + \text{Time}(Q)$ or
  - $\text{Time}(P; Q) = \Theta(max(Time(P), Time(Q)))$
- **for** loop: **for** $i = 1$ to m **do** $P(i)$
  - $t(i)$ is the time complexity of $P(i)$
  - time complexity of the **for** loop is $\sum_{i=1}^{m} t(i)$

Analysis of algorithms

while (**repeat**) loop

- Specify a function of variables of the loop such that this function reduces during the loop
- To evaluate the running time, we analyze how the function reduces during the loop

Example: binary search

Function BinarySearch($T[1..n], x$)

begin

$i \leftarrow 1$; $j \leftarrow n$;

while $i < j$ **do**

$k \leftarrow (i + j)/2$;

case

$x < T[k]$: $j \leftarrow k - 1$;

$x = T[k]$: $i \leftarrow k$; $j \leftarrow k$; exit;

$x > T[k]$: $i \leftarrow k + 1$;

endcase

endwhile

end

Example: binary search
Denote

- $d = j - i + 1$ (number of elements of the array to be investigated)
- i^*, j^*, d^* respectively the values of i, j, d after a loop

We have

- If $x < T[k]$ then $i^* = i$, $j^* = (i + j)/2 - 1$, $d^* = j^* - i^* + 1 \leq d/2$
- If $x > T[k]$ then $j^* = j$, $i^* = (i + j)/2 + 1$, $d^* = j^* - i^* + 1 \leq d/2$
- If$x = T[k]$ then $d^* = 1$

Hence, the number of iterations of the loop is $\lceil logn \rceil$

Master theorem

$T(n) = aT(n/b) + cn^k$ with $a \geq 1, b > 1, c > 0$ are constant

- If $a > b^k$, then $T(n) = \Theta(n^{log_b a})$
- If $a = b^k$, then $T(n) = \Theta(n^k logn)$ with $log\, n = log_2 n$
- If $a < b^k$, then $T(n) = \Theta(n^k)$

Example

- $T(n) = 3T(n/4) + cn^2 \Rightarrow T(n) = \Theta(n^2)$
- $T(n) = 2T(n/2) + n^{0.5} \Rightarrow T(n) = \Theta(n)$
- $T(n) = 16T(n/4) + n \Rightarrow T(n) = \Theta(n^2)$
- $T(n) = T(3n/7) + 1 \Rightarrow T(n) = \Theta(logn)$

Sorting

- Put elements of a list in a certain order
- Designing efficient sorting algorithms is very important for other algorithms (search, merge, etc.)
- Each object is associated with a key and sorting algorithms work on these keys.
- Two basic operations that used mostly by sorting algorithms
  - Swap(a, b): swap the values of variables a and b
  - Compare(a, b): return
    - true if a is before b in the considered order
    - false, otherwise.
- Without loss of generality, suppose we need to sort a list of numbers in nondecreasing order

- A sorting algorithm is called **in-place** if the size of additional memory required by the algorithm is $\mathcal{O}(1)$ (which does not depend on the size of the input array)
- A sorting algorithm is called **stable** if it maintains the relative order of elements with equal keys
- A sorting algorithm uses only comparison for deciding the order between two elements is called **Comparison-based sorting algorithm**

Insertion Sort

- At iteration k, put the k^{th} element of the original list in the right order of the sorted list of the first k elements ($\forall k = 1, \ldots, n$)
- Result: after k^{th} iteration, we have a sorted list of the first k^{th} elements of the original list

```
void insertion_sort(int a[], int n){
  int k;
  for(k = 2; k <= n; k++){
    int last = a[k];
    int j = k;
    while(j > 1 && a[j-1] > last){
      a[j] = a[j-1];
      j--;
    }
    a[j] = last;
  }
}
```

Selection Sort

- Put the smallest element of the original list in the first position
- Put the second smallest element of the original list in the second position
- Put the third smallest element of the original list in the third position
- ...

```
void selection_sort(int a[], int n){
  for(int k = 1; k <= n; k++){
    int min = k;
    for(int i = k+1; i <= n; i++)
      if(a[min] > a[i])
        min = i;
    swap(a[k],a[min]);
  }
}
```

Bubble sort

- Pass from the beginning of the list: compare and swap two adjacent elements if they are not in the right order
- Repeat the pass until no swaps are needed

```
void bubble_sort(int a[], int n){
  int swapped;
  do{
    swapped = 0;
    for(int i = 1; i < n; i++)
    if(a[i] > a[i+1]){
      swap(a[i],a[i+1]);
      swapped = 1;
    }
  }while(swapped == 1);
}
```

Merge sort

Divide-and-conquer

- Divide the original list of $n/2$ into two lists of $n/2$ elements
- Recursively merge sort these two lists
- Merge the two sorted lists

Merge sort

```
void merge(int a[], int L, int M, int R){
  // merge two sorted list a[L..M] and a[M+1..R]
  int i = L;// first position of the first list a[L
  int j = M+1;// first position of the second list
  for(int k = L; k <= R;k++){
    if(i > M){// the first list is all scanned
      TA[k] = a[j]; j++;
    }else if(j > R){// the second list is all scann
      TA[k] = a[i]; i++;
    }else{
      if(a[i] < a[j]){
        TA[k] = a[i]; i++;
      }else{
        TA[k] = a[j]; j++;
      }
    }
  }
  for(int k = L; k <= R; k++)
```

Merge sort

```
void merge_sort(int a[], int L, int R){
  if(L < R){
    int M = (L+R)/2;
    merge_sort(a,L,M);
    merge_sort(a,M+1,R);
    merge(a,L,M,R);
  }
}
```

Quick sort

- Pick an element, called a **pivot**, from the original list
- Rearrange the list so that:
  - All elements less than **pivot** come before the **pivot**
  - All elements greater or equal to to **pivot** come after **pivot**
- Here, **pivot** is in the right position in the final sorted list (it is fixed)
- Recursively sort the sub-list before **pivot** and the sub-list after **pivot**

Quick sort

```
void quick_sort(int a[], int L, int R){
  if(L < R){
   int index = (L+R)/2;
    index = partition(a,L,R,index);
    if(L < index)
      quick_sort(a,L,index-1);
    if(index < R)
      quick_sort(a,index+1,R);
  }
}
```

Quick sort

```
int partition(int a[], int L, int R, int indexPivot
  int pivot = a[indexPivot];
  swap(a[indexPivot],a[R]);// put the pivot in the
  int storeIndex = L; // store the right position o

  for(int i = L; i <= R-1; i++){
    if(a[i] < pivot){
      swap(a[storeIndex],a[i]);
      storeIndex++;
    }
  }
  swap(a[storeIndex],a[R]); // put the pivot in the

  return storeIndex;
}
```

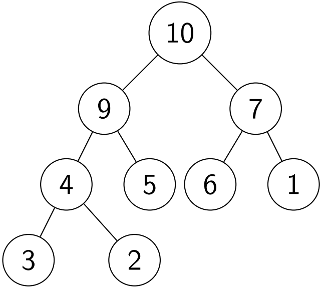
Heap sort

Sort a list $A[1..N]$ in nondecreasing order

1. Build a heap out of $A[1..N]$
2. Remove the largest element and put it in the N^{th} position of the list
3. Reconstruct the heap out of $A[1..N-1]$
4. Remove the largest element and put it in the $N-1^{th}$ position of the list
5. ...

Heap sort - Heap structure

- Shape property: Complete binary tree with level L
- Heap property: each node is greater than or equal to each of its children (max-heap)



1	2	3	4	5	6	7	8	9
10	9	7	4	5	6	1	3	2

Heap sort

- Heap corresponding to a list $A[1..N]$
  - Root of the tree is $A[1]$
  - Left child of node $A[i]$ is $A[2 * i]$
  - Right child of node $A[i]$ is $A[2 * i + 1]$
  - Height is $logN + 1$
- Operations
  - Build-Max-Heap: construct a heap from the original list
  - Max-Heapify: repair the following binary tree so that it becomes Max-Heap
    - A tree with root $A[i]$
    - $A[i] < max(A[2 * i], A[2 * i + 1])$: heap property is not hold
    - Subtrees rooted at $A[2 * i]$ and $A[2 * i + 1]$ are Max-Heap

Heap sort

```
void heapify(int a[], int i, int n){
  // array to be heapified is a[i..n]
  int L = 2*i;
  int R = 2*i+1;
  int max = i;
  if(L <= n && a[L] > a[i])
    max = L;
  if(R <= n && a[R] > a[max])
    max = R;
  if(max != i){
    swap(a[i],a[max]);
    heapify(a,max,n);
  }
}
```

Heap sort

```
void buildHeap(int a[], int n){
    // array is a[1..n]
  for(int i = n/2; i >= 1; i--){
    heapify(a,i,n);
  }
}

void heap_Sort(int a[], int n){
    // array  is a[1..n]
  buildHeap(a,n);
  for(int i = n; i > 1; i--){
    swap(a[1],a[i]);
    heapify(a,1,i-1);
  }
}
```

Data structures

- List, Stack, Queue
- Graphs
- Trees

List

- Collection of objects which are arranged in a linear order
- Array
  - Continuous allocation
  - Accessing elements via indices
- Linked List
  - Elements are not necessarily allocated continuously
  - User pointer to link an element with its successor
  - Accessing elements via pointers

Stacks

- An ordered list in which all insertions and deletions are made at one end (called top)
- Principle: the last element inserted into the stack must be the first one to be removed (**Last-In-First-Out**)
- Operations
  - Push(x, S): push an element x into the stack S
  - Pop(S): remove an element from the stack S, and return this element
  - Top(S): return the element at the top of the stack S
  - Empty(S): return true if the stack S is empty

Queues

- An ordered list in which the insertions are made at one end (called tail) and the deletions are made at the other end (called head)
- Principle: the first element inserted into the queue must be the first one to be removed (**First-In-First-Out**)
- Applications: items do not have to be processed immediately but they have to be processed in FIFO order
  - Data packets are stored in a queue before being transmitted over the internet
  - Data is transferred asynchronously between two processes: IO buffered, piples, etc.
  - Printer queues, keystroke queues (as we type at the keyboard), etc.

Queues

- Operations
  - Enqueue(x, Q): push an element x into the queue Q
  - Dequeue(Q): remove an element from the queue Q, and return this element
  - Head(Q): return the element at the head of the queue Q
  - Tail(Q): return the element at the tail of the queue Q
  - Empty(Q): return true if the queue Q is empty

Java Libraries

- List
  - ArrayList (dynamic array): get(int index), size(), remove(int index), add(int index, Object o), indexOf(Object o)
  - LinkedList (doubly linked list): remove, poll, element, peek, add, offer, size
- Stack
  - push, pop, size
- Queue
  - LinkedList
  - remove, poll, element, peek, add, offer, size
- Set
  - Collection of items
  - Methods: add, size, contains
- Map
  - Map an object (key) to another object (value)
  - Methods: put, get, keySet

Examples

```java
package week2;

import java.util.ArrayList;

public class ExampleArrayList {

  public ExampleArrayList(){
    ArrayList<Integer> L = new ArrayList();
    for(int i = 1; i <= 10; i++)
      L.add(i);
    for(int i = 0; i < L.size(); i++){
      int item = L.get(i);
      System.out.print(item + " ");
    }
    System.out.println("size of L is " + L.size());
  }
  public static void main(String[] args) {
    ExampleArrayList EAL = new ExampleArrayList();
  }
}
```

Examples

```java
package week2;
import java.util.HashSet;
public class ExampleSet {

  public void test(){
    HashSet<Integer> S = new HashSet();
    for(int i = 1; i <= 10; i ++)
      S.add(i);
    for(int i: S){
      System.out.print(i + " ");
    }
    System.out.println("S.size() = " + S.size());
System.out.println(S.contains(20));
  }
  public static void main(String[] args) {
    ExampleSet ES = new ExampleSet();
    ES.test();
  }
}
```

Examples

```
package week2;
import java.util.LinkedList;
import java.util.Queue;
public class ExampleQueue {
  public static void main(String[] args) {
    Queue Q = new LinkedList();
    /*
     * Q.element(): return the head of the queue without removing it.
     * If Q is empty, then raise exception
     * Q.peek(): return the head of the queue without removing it.
     * If Q is empty, then return null
     * Q.remove(): remove and return the head of the queue.
     * If Q is empty, then raise exception
     * Q.poll(): remove and return the head of the queue.
     * If Q is empty, then return null
     * Q.add(e): add an element to the tail of Q.
     * If no space available, then raise exception
     * Q.offer(e): add an element to the tail of Q. If no space availa
     * then return false. Otherwise, return true
     */
    for(int i = 1; i <= 10; i++)  Q.offer(i);
    while(Q.size() > 0){
      int x = (int)Q.remove();
      System.out.println("Remove " + x + ", head of Q is " + Q.peek()
    }
  }
```

Examples

```java
package week2;

import java.util.*;
public class ExampleStack {
  public ExampleStack(){
    /*
     * S.push(e): push an element to the stack
     * S.pop: remove the element at the top of the stack and return i
     */
    Stack S = new Stack();
    for(int i = 1; i <= 10; i++)
      S.push(i + "000");
    while(S.size() > 0){
      String x = (String)S.pop();
      System.out.println(x);
    }
  }
  public static void main(String[] args) {
    ExampleStack S = new ExampleStack();
  }
}
```

Examples

```java
package week2;

import java.util.HashMap;
public class ExampleHashMap {

  public ExampleHashMap(){
    HashMap<String, Integer> m = new HashMap<String, Integer>();
    m.put("abc",1);
    m.put("def", 1000);
    m.put("xyz", 100000);
    for(String k: m.keySet()){
      System.out.println("key = " + k + " map to " + m.get(k));
    }
  }
  public static void main(String[] args) {
    ExampleHashMap EHM = new ExampleHashMap();
  }
}
```

Examples

```java
package week2;
import java.util.Scanner;
import java.util.HashMap;
import java.io.File;
public class CountWords {
  public CountWords(String filename){
    HashMap<String, Integer> count = new HashMap<String, Integer>();
    try{
      Scanner in = new Scanner(new File(filename));
      while(in.hasNext()){
        String s = in.next();
        if(count.get(s) == null)
        count.put(s, 0);
        count.put(s, count.get(s) + 1);
      }
      for(String w: count.keySet())
        System.out.println("Word " + w + " appears " + count.get(w) +
      in.close();
    }catch(Exception ex){
      ex.printStackTrace();
    }
  }
  public static void main(String[] args) {
    CountWords  CW = new CountWords("data\\week2\\CountWords.txt");
  }
}
```

Checking parentheses expression

```
private boolean match(char c, char cc){
  if(c == '(' && cc == ')') return true;
  if(c == '[' && cc == ']') return true;
  if(c == '{' && cc == '}') return true;
  return false;
}
public boolean check(String expr){
  Stack S = new Stack();
  for(int i = 0; i < expr.length(); i++){
    char c = expr.charAt(i);
    if(c == '(' || c == '{'|| c == '[')
      S.push(c);
    else{
      if(S.size() == 0) return false;
      char cc = (char)S.pop();
      if(!match(cc,c)) return false;
    }
  }
  return S.size() == 0;
}
```

Water Jug Problem

There are two jugs, a a-gallon one and a b-gallon one (a, b are positive integer). There is a pump with unlimited water. Neither jug has any measuring marking on it. How can you get exactly c-gallon jug (c is a positive integer)?

Water Jug Problem

- Search problem
- State (x, y): quantity of water in two jugs
- Neighboring states
  - $(x, 0)$
  - $(0, y)$
  - (a, y)
  - (x, b)
  - $(a, x + y - a)$ if $x + y >= a$
  - $(x + y, 0)$ if $x + y < a$
  - $(x + y - b, b)$ if $x + y >= b$
  - $(0, x + y)$ if $x + y < b$
- Final states: (c, y) or (x, c)

Water Jug Problem

Algorithm 1: WaterJug(a, b, c)

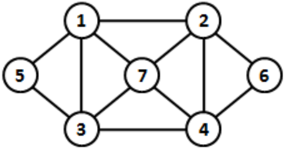
$Q \leftarrow \emptyset$;
Enqueue((a, b), Q);
while *Q is not empty* **do**
    (x, y) $\leftarrow$ Dequeue(Q);
    foreach $(x', y') \in$ *neighboring states of* (x, y) **do**
        if $(x' = c \vee y' = c)$ **then**
            Solution();
            BREAK;
        else
            Enqueue((x', y'), Q);

Graphs and Trees

- Many objects in our daily lives can be modelled by graphs
  - Internets, social networks (facebook), transportation networks, biological networks, etc.
- An graph G is a mathematical object consisting two finites sets, $G = (V, E)$
  - V is the set of vertices
  - E is the set of edges connecting these vertices
- Graphs have many types: directed, undirected, multigraphs, etc.

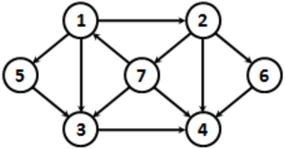
Definitions

- An undirected graph $G = (V, E)$
  - $V = (v_1, v_2, \ldots, v_n)$ is the set of vertices or nodes
  - $E \subseteq V \times V$ is the set of edges (also called undirected edges). E is the set of unordered pair (u, v) such that $u \neq v \in V$
  - $(u, v) \in E$ iff $(v, u) \in E$



Definitions

- A directed graph $G = (V, E)$
  - $V = (v_1, v_2, \dots, v_n)$ is the set of vertices or nodes
  - $E \subseteq V \times V$ is the set of arcs (also called directed edges). E is the set of ordered pair (u, v) such that $u \neq v \in V$



Definitions

- Given a graph $G = (V, E)$, for each $(u, v) \in E$, we say u and v are adjacent
- Given an undirected graph $G = (V, E)$
  - degree of a vertex v is the number of edges connecting it: $deg(v) = \sharp\{(u, v) \mid (u, v) \in E\}$
- Given a directed graph $G = V, E)$
  - An incoming arc of a vertex is an arc that enters it
  - An outgoing arc of a vertex is an arc that leaves it
  - indegree (outdegree) of a vertex v is the number of its incoming (outgoing) arcs

$$deg^{+}(v) = \sharp\{(v, u) \mid (v, u) \in E\}, deg^{-}(v) = \sharp\{(u, v) \mid (u, v) \in E\}$$

Definitions

Theorem

Given an undirected graph $G = (V, E)$, we have

$$2 \times |E| = \sum_{v \in V} deg(v)$$

Theorem

Given a directed graph $G = (V, E)$, we have

$$\sum_{v \in V} deg^{+}(v) = \sum_{v \in V} deg^{-}(v) = |E|$$

Definition - Paths, cycles

- Given a graph $G = (V, E)$, a path from vertex u to vertex v in G is a sequence $\langle u = x_0, x_1, \ldots, x_k = v \rangle$ in which $(x_i, x_{i+1}) \in E), \forall i = 0, 1, \ldots, k - 1$
  - u: starting point (node)
  - v: terminating point
  - k is the length of the path (i.e., number of its edges)
- A cycle is a path such that the starting and terminating nodes are the same
- A path (cycle) is called simple if it contains no repeated edges (arcs)
- A path (cycle) is called elementary if it contains no repeated nodes

Connectivity

- Given an undirected graph $G = (V, E)$. G is called **connected** if for any pair (u, v) $(u, v \in V)$, there exists always a path from u to v in G
- Given a directed graph $G = (V, E)$, G is called
  - **weakly connected** if the corresponding undirected graph of G (i.e., by removing orientation on its arcs) is connected
  - **strongly connected** if for any pair (u, v) $(u, v \in V)$, there exists always a path from u to v in G
- Given an undirected graph $G = (V, E)$
  - an edge e is called **bridge** if removing e from G increases the number of connected components of G
  - a vertex v is called **articulation point** if removing it from G increases the number of connected components of G

Connectivity

Theorem

An undirected connected graph G can be oriented (each edge of G is oriented) to obtain a strongly connected graph iff each edge of G lies on at least one cycle

Planar graphs - Euler Polyhedron Formula

Theorem

Given a connected planar graph having n vertices, m edges. The number of regions divided by G is $m - n + 2$.

Planar graphs - Kuratowski's theorem

Definition

A **subdivision** of a graph G is a new graph obtained by replacing some edges by paths using new vertices, edges (each edge is replaced by a path)

Theorem

Kuratowski *A graph G is planar iff it does not contain a subdivision of $K_{3,3}$ or K_5*

Graph representation

- Two standard ways to represent a graph $G = (V, E)$
  - Adjacency list
    - Appropriate with sparse graphs
    - $Adj[u] = \{v \mid (u, v) \in E\}, \forall u \in V$
  - Adjacency matrix
    - Appropriate with dense graphs
    - $A = (a_{ij})_{n \times n}$ such that (suppose $V = \{1, 2, \ldots, n\}$)

$$a_{ij} = \begin{cases} 1 & \text{if } (i, j) \in E, \\ 0 & \text{otherwise} \end{cases}$$

Graph representation

- In some cases, we can use incidence matrix to represent a directed graph $G = (V, E)$

$$b_{ij} = \begin{cases} -1 & \text{if edge } j \text{ leaves vertex } i, \\ 1 & \text{if edge } j \text{ enters vertex } i, \\ 0 & \text{otherwise} \end{cases}$$

Applications of Graphs and Trees

- Social netwokrs analysis
- Transportation networks
- Telecommunication networks
- etc.