Fundamentals of information systems

Davide Volpi

January 13, 2025

Contents

1 Introduction

1.1 What is an IS?

Is a set of interrelated elements that collect, manipulate, store and disseminate data and information, and that provide a corrective reaction to meet an objective.

1.2 Technology Infrastructure

We need to know how to collect, manipulate, store and process data into information through software, database and networks.

2 Algorithmic thinking

2.1 Introduction to algorithmic thinking

The 4 pillars of algorithmic thinking are:

1. **Decomposition:** split the problem in tasks.
2. **Pattern matching:** apply a method to a different application from the basic one.
3. **Generalization:** make a specific method less specific and useful for the task.
4. **Abstraction:** imagine the functioning of the algorithm to estimate the computational time (time+effort).

2.2 Design and Analysis of Algorithms

Algorithms: sequences of basic steps a non-intelligent being can blindly follow to solve a problem. Is a well-defined computational procedure that takes some values as input and produce some values as output. Algorithms give you a language to talk about problems and solutions. The algorithm must respect:

1. **Finiteness:** must always terminate after a finite number of steps.
2. **Definitness:** Each step must be precisely defined.
3. **Input:** 0 or more.
4. **Output:** 1 or more.
5. **Effectiveness:** operations must all be sufficiently basic that they can in principle be done exactly in a finite amount of time by someone using pencil and paper.

We can define algorithm by using **pseudo-code** or any available **programming language**.
An algorithm is **correct** if it halts with the correct output, so we say that an algorithm is correct if it **solves** the given computational problem.
Computational problem: Π is a mathematical relation between a set I of possible instances and a set S of possible solutions: $\Pi \subseteq I \times S$ such that $\forall i \in I$ there exists at least one solution s $\in$ S such that $(i, s) \in \Pi$. An algorithm A solves the problem $\Pi \subseteq I \times S$ if $\forall i \in I,\ A(i) \rightarrow s$.

2.2.1 The study of performance

There are a lot of things more important than performance such as cost, maintainability, security... but often performances define the line between **feasible/unfeasible** and is also a **measurable value**.

2.2.2 Algorithmic analysis

It is used to determinate **how good** an algorithm is. The idea is to take an algorithm and to determinate its **quantitative behaviour**. The core is to determinate the **performance characteristics** of an algorithm. **Efficiency** is often measured in terms of the input size (n).

3 Sorting Algorithms

3.1 Insertion Sort

The leftmost element w.r.t. the current index is always ordered.
When we move an element from j to $i < j$, we have to make room for it by shifting all the elements from i to $j - 1$.

```
INSERTION-SORT(A)
for j = 2 to length(A) do
    key = A[j]
    i = j - 1
    while i > 0 and A[i] > key do
        A[i + 1] = A[i]
        i = i - 1
    A[i + 1] = key
```

```
def insertionSort(A):
   for j in range(1, len(A)):
        key = A[j]
        i = j
        while i > 0 and A[i-1] > key:
            A[i] = A[i-1]
            i = i - 1
        A[i] = key
```

3.1.1 Computational costs

Best case analysis: the sequence is just sorted

- $T(n) = c_1 n + c_2(n-1) + c_4(n-1) + c_5 \sum_{j=2}^{n} t_j + (c_6 + c_7) \sum_{j=2}^{n} (t_j - 1) + c_8(n-1) =$
- $T(n) = n + (n-1) + (n-1) + \sum_{j=2}^{n} t_j + 2 \sum_{j=2}^{n} (t_j - 1) + (n-1) =$
- $T(n) = n + (n-1) + (n-1) + (n-1) + 0 + (n-1) = 5n - 4 = \mathbf{bn + c}$

Worst case analysis: decreasing ordered

- $T(n) = c_1 n + c_2(n-1) + c_4(n-1) + c_5 \sum_{j=2}^{n} t_j + (c_6 + c_7) \sum_{j=2}^{n} (t_j - 1) + c_8(n-1) =$
- $T(n) = n + (n-1) + (n-1) + c_5(\frac{n(n+1)}{2} - 1) + (c_6 + c_7)(\frac{n(n+1)}{2}) + c_8(n-1) = \mathbf{an^2 + bn + c}$

3.2 Selection Sort

- **Step 1:** Scan the list and get the min.
- **Step 2:** Swap the min with the first element of the list.
- **Step 3:** Repeat from the next element until the length-1 element.

```
SELECTION-SORT(A)
n=length[A]
for j=1 to n-1 do
    smallest = j
    for i = j+1 to n do
        if A[i]<A[smallest] then
            smallest = i
    exchange A[j] with A[smallest]
```

3.2.1 Computational costs

Best case analysis = Worst case analysis: there is no best or worst case scenario, is the same

- $T(n) = c_1 + c_2 n + c_3(n-1) + c_4 \sum_{j=1}^{n-1} t_j + (c_5 + c_6) \sum_{j=1}^{n-1} (t_j - 1) + c_7(n-1)$
- $T(n) = 3n - 1 + \frac{n(n-1)}{2} + n(n-1) - 1 = \mathbf{an^2 + bn + c}$

This algorithm complexity is always $\mathbf{n^2}$.

3.3 Merge Sort

- **Step 1:** Assume we have an unordered sequence of n elements.
- **Step 2:** Divide the sequence in two sequences with length n/2.
- **Step 3:** Sort the two sequences using merge-sort recursively.
- **Step 4:** Merge the two ordered sequences into the output sequence.

```
MERGE-SORT(A,p,r)
if(p<r)
    q = (p + r) // 2 #floor division
    MERGE-SORT(A,p,q)
    MERGE-SORT(A,q+1,r)
    MERGE(A,p,q,r)
```

3.3.1 Computational costs

- The complexity of MERGE is $T(n) = \Theta(n)$
- The general complexity of the algorithm is: $\begin{cases} T(n) = \Theta(n) & \text{if } n \leqslant c \\ T(n) = aT\left(\frac{n}{2}\right) + \Theta(n) & \text{otherwise} \end{cases}$
- The best case scenario complexity is $T(n) = \Theta(n)$
- The worst case scenario complexity is $T(n) = \Theta(nlog(n))$

3.4 Heap Sort

We need to sort an array, so we build a heap from this array with Build-Max-Heap. Then we exchange $A[i]$ with $A[i]$, going from $A.length$ down to 2 and we do $heap - size = heap - size - 1$. Then we call $Max - Heapify(A, 1)$.

```
Input: A is an array
HEAPSORT(A)
BUILD-MAX-HEAP(A)
for i=len(A) downto 2 do
    exchange A[1] with A[i]
    A.heap-size = A.heap-size-1
    MAX-HEAPIFY(A,1)
```

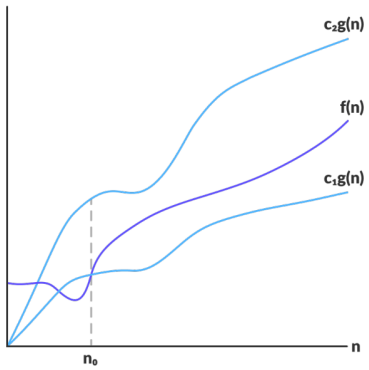
3.4.1 Computational costs

- The complexity is $O(n) + O((n-1)(log(n))) = O(nlog(n))$.
- Is an in-place algorithm.

4 Asymptotical Analysis

Say the input has size n then we care about how the algorithm performs when n $\rightarrow \infty$. An algorithm A which is asymptotically better than an algorithm B will perform better for all, but very small inputs. An algorithm A which is asymptotically better than an algorithm B will perform better for all, but very small inputs ($n \geqslant n_0$ where n_0 is a small value selected ad-hoc).
We can define:

- $\Theta g(n) = \{f(n) : \exists c_1, c_2 > 0 \wedge n_0\, 0 \text{ s.t. } 0 \leqslant c_1 g(n) \leqslant f(n) \leqslant c_2 g(n), \forall n \geqslant n_0\}$
  - $g(n)$ defines the upper and lower bounds of $f(n)$.



- $Og(n) = \{f(n) : \exists c > 0 \wedge n_0\, 0 \text{ s.t. } 0 \leqslant f(n) \leqslant cg(n), \forall n \geqslant n_0\}$
  - $\Theta(g(n)) \in O(g(n))$
  - This is the worst case scenario
- $\Omega g(n) = \{f(n) : \exists c > 0 \wedge n_0\, 0 \text{ s.t. } 0 \leqslant cg(n) \leqslant f(n), \forall n \geqslant n_0\}$
  - This is the best case scenario

For any two functions $f(n)$ and $g(n)$, we have $f(n) = \Theta(g(n))$ if $f(n) = O(g(n)) \wedge f(n) = \Omega(g(n))$

5 Recursion

5.1 Divide and Conquer

An algorithm is **Recursive** if it calls itself one or more times to solve a problem. Is useful when a task can be split into similar, but smaller subtasks. The algorithm calls itself until it reaches his base case.
When an algorithm calls itself you have a nested call that creates an **Execution Stack** where the current execution is paused, the execution context associated with the current algorithm execution is stored in the stack and the nested call executes. After it ends, the previous call is retrieved from the stack and the execution is resumed from where it stopped.
These algorithm typically follow a **Divide and Conquer Strategy**:

- **Divide** divide the problem into smaller sub-problems.
- **Conquer** conquer the sub-problems by solving them one at one.
- **Combine** the solutions of the sub-problems into the final solution.

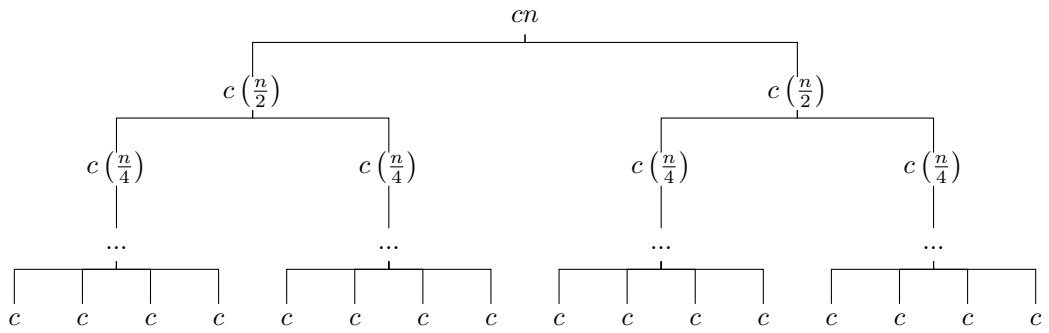
Some examples of Divide and Conquer algorithms are:

- Binary search
- MergeSort

5.2 The Recursion Tree

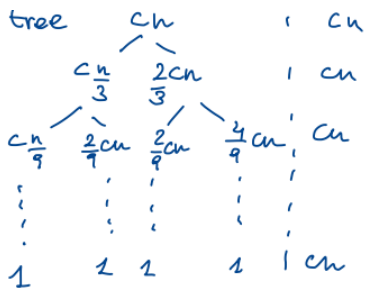
Given the equation $\begin{cases} T(n) = \Theta(n) & \text{if } n \leqslant c \\ T(n) = aT\left(\frac{n}{2}\right) + \Theta(n) & \text{otherwise} \end{cases}$ we can build the recursion tree as follow:

- For the algorithm we consider n as power of 2.
- The size of the sub-problems is floor$(\frac{n}{2}) = \frac{n}{2}$



The tree has $\log_2(n) + 1$ levels and each level costs cn. So the final complexity is $n \log_2(n)$ because we have to compute the base case $\log_2(n) + 1$ times.
What is the complexity of $T(n) = 3T(\lfloor \frac{n}{4} \rfloor) + \Theta(n^2)$?

- $T(n) = 3T(\frac{n}{4}) + \Theta(n^2)$
- $T(n) = 3T(\frac{n}{4}) + cn^2$ for $c > 0$
- 3 is the **number of subproblems = a** and 4 is **b**
  - In general, $T(n) = aT(\frac{n}{b}) + \Theta(n^2)$ the **height** is $h = \log_b(n)$ and **# of leaves** is $n^{\log_b(a)}$.
- $h = \log_4(n)$ and the number of leaves is $n^{\log_4(3)}$
- The final equation is $T(n) = \sum_{i=0}^{\log_4(n)-1} (\frac{3}{16})^i$
- We need to use the **geometric series formula** that states having $\sum_{i=0}^{n} p^i$
  - $T(n) = \frac{p^(n+1)-1}{p-1}$ if $p \geqslant 1$
  - $T(n) < \frac{1}{1-p}$ if $p < 1$
- In conclusion we have $T(n) < \frac{1}{1-p} c_1 n^2 + n^{\log_4(3)} = On^2$



5.3 Binary Search

- This is the simplest way to find if an element is into an ordered list.
- The recursive equation is $T(n) = T(\frac{n}{2}) + \Theta(1)$ and this mean the complexity is $\Theta(\log n)$.

```
ITERATIVE-BINARY-SEARCH(S,x,low,high)
    while low <= high
        mid=floor[(high+low)/2]
        if S[mid] == x then
            return mid
        else if S[mid] < x then
            low = mid + 1
        else if S[mid] > x then
        high = mid - 1
    return NULL
```

5.4 Matrix Multiplications

Given two matrices A and B (rows * columns) determine $C = A * B$:

1. Divide the matrices $A_(n*n) = \begin{bmatrix} A11 & A12 \\ A21 & A22 \end{bmatrix} B_(n*n) = \begin{bmatrix} B11 & B12 \\ B21 & B22 \end{bmatrix} C_(n*n) = \begin{bmatrix} C11 & C12 \\ C21 & C22 \end{bmatrix}$
   where we have that $\begin{array}{ll} C11 = & A11*B11 + A12*B21 \\ C12 = & A11*B12 + A12*B22 \\ C21 = & A21*B11 + A22*B21 \\ C22 = & A21*B12 + A22*B22 \end{array}$, there are 8 matrix multiplications.
2. The recursion equation is $\begin{cases} T(n) = \Theta(1) & \text{if } n = 1 \\ T(n) = 8T(\frac{n}{2}) + \Theta(n^2) & \text{if} n > 1 \end{cases} = \Theta(n^3)$

```
SMMR(A,B)
    n = A.rows
    let C be a bew n * n matrix
    if n == 1
        C_11 = A_11 * B_11
    else
        partition A,B,C as shown
        \\ this can be done by copying entries
        \\ with Theta(n^2) complexity
        \\ by using index partitioning
        C_11 = SNNR(A_11,B_11) + SNNR(A_12,B_21)
        C_12 = SNNR(A_11,B_12) + SNNR(A_12,B_22)
        C_21 = SNNR(A_21,B_11) + SNNR(A_22,B_21)
        C_11 = SNNR(A_21,B_12) + SNNR(A_22,B_22)
    retrurn C
```

5.4.1 Strassen Algorithm

1. Divide the matrices A and B as in the SMMR case.
2. Create 10 matrices $S_1, ..., S_10$ each of which is $\frac{n}{2} * \frac{n}{2}$ and they are the sum or th difference of the matrices created in step 1 $\rightarrow \Theta(n^2)$ (subtract the columns and sum the rows).
3. Using the submatrices created in step 1 and step 2, recursively compute 7 matrices that are the products $P_1, ..., P_7$ each of which is $\frac{n}{2} * \frac{n}{2} \rightarrow \Theta(n^{log7}) = \Theta(n^{2.8073})$.
4. Compute the final submatrices $C_11, C_12, C_21, C_22 \rightarrow \Theta(n^2)$.

6 Master Method for Solving Recurrences

Theorem: Let $a \geqslant 1$ and $b > 1$ be constants and $f(n)$ be a function and let $T(n) = aT(\frac{n}{b}) + f(n)$ where we interpret $\frac{n}{b}$ as $\lfloor \frac{n}{b} \rfloor$ or $\lceil \frac{n}{b} \rceil$. Then, $T(n)$ has the following bounds:

- If $f(n) = O(n^{log_b(a-\epsilon)})$ with $\epsilon > 0$, then $T(n) = \Theta(n^{log_b(a)})$.
  - $n^{log_b(a)}$ is larger than $f(n)$ because we can equalize $f(n)$ to $n^{log_b(a)}$ by subtracting a possible $\epsilon > 0$.
- If $f(n) = \Theta(n^{log_b(a)})$, then $T(n) = \Theta(n^{log_b(a)} log(n))$.
  - The two functions are the same $\rightarrow T(n) = \Theta(n^{log_b(a)} log(n))$.
- If $f(n) = \Omega(n^{log_b(a+\epsilon)})$ with $\epsilon > 0$ and if $af(\frac{n}{b}) \leqslant cf(n)$ for some $c < 1$ and all sufficiently large n, then $T(n) = \Theta(f(n))$.
  - $f(n)$ is larger than $n^{log_b(a)} \rightarrow T(n) = \Theta(f(n))$.

6.1 Proof of the Theorem

6.1.1 Part 1 of the Theorem: Exact powers

Given the general recurrence $T(n) = aT(\frac{n}{b}) + f(n)$ we extract 3 lemmas:

1. Reduces the recurrence to an equation that contains a summation.
2. Provides a bound to this summation.
3. Puts the 2 lemmas together and proves the Master Theorem for exact powers of b.

Lemma 1
Let $a \geqslant 1$ and $b > 1$ be constants, and let $f(n)$ be a non negative function defined on exact powers of b. Define $T(n)$ by the recurrence $\begin{cases} \Theta(1) & \text{if } n = 1 \\ aT(\frac{n}{b}) + f(n) & \text{if } n^i, i \in N \end{cases}$ Then, $T(n) = \Theta(n^{log_b a}) + \sum_{j=0}^{log_b(n-1)} a^i f(\frac{n}{b^j})$

Lemma 2
Let $a \geqslant 1$ and $b > 1$ be constants, and let $f(n)$ be a non negative function defined on exact powers at b. A function $g(n)$ defined over exact powers of b by $g(n) = \sum_{j=0}^{log_b(n-1)} a^j f(\frac{n}{b^j})$. It can be bound asymptotically as follows:

1. If $f(n) = O(n^{log_b(a-\epsilon)}, \epsilon > 0$ then $g(n) = O(n^{log_b a})$.
2. If $f(n) = \Theta(n^l og_b a)$, then $g(n) = \theta(n^{log_b a} log(n))$.
3. If $f(\frac{n}{b}) \leqslant cf(n), c > 1$ and $\forall n \geqslant b$, then $g(n) = \Theta(g(n))$

Lemma 3
Let $a \geqslant 1$ and $b > 1$ be constants, and let $f(n)$ be a non negative function defined on exact powers at b. Define $T(n)$ by the recurrence $\begin{cases} \Theta(1) & \text{if } n = 1 \\ aT(\frac{n}{b}) + f(n) & \text{if } n^i, i \in N \end{cases}$ Then, $T(n)$ can be bound asymptotically as follows

1. If $f(n) = O(n^{log_b(a-\epsilon)}$ for some $\epsilon > 0$.
2. If $f(n) = \Theta(n^{log_b a}$, then $T(n) = \Theta(n^{log_b a} log(n))$.
3. If $f(n) = \Omega(n^{log_b(a+\epsilon)}, \epsilon > 0$ and if $af(\frac{n}{b} \leqslant cf(n), c < 1$ then $T(n) = \Theta(f(n))$.

6.1.2 Part 2 of the Theorem: Floors and Ceilings

When the recurrence involves floors or ceilings, i.e., $T(n) = aT(\lfloor n/b \rfloor) + f(n)$ or $T(n) = aT(\lceil n/b \rceil) + f(n)$, the key observation is that the impact of rounding in $\lfloor n/b \rfloor$ or $\lceil n/b \rceil$ is negligible for asymptotic analysis, as it introduces at most a constant factor difference in each term of the recurrence. Thus, the asymptotic behavior remains governed by the same balance between $f(n)$ and $n^{\log_b a}$:

- If $f(n) = O(n^{\log_b(a)-\epsilon})$ for some $\epsilon > 0$, then $T(n) = \Theta(n^{\log_b a})$.
- If $f(n) = \Theta(n^{\log_b a})$, then $T(n) = \Theta(n^{\log_b a} \log n)$.
- If $f(n) = \Omega(n^{\log_b(a)+\epsilon})$ for some $\epsilon > 0$ and $af(\lfloor n/b \rfloor) \leqslant cf(n)$ for some $c < 1$, then $T(n) = \Theta(f(n))$.

7 Base Data Structures

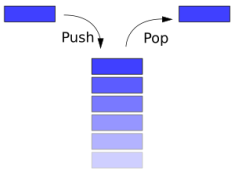
ADT(Abstract Data Type)= It's a mathematical model of a data structure that it specifies the type of the data stored, the supported operations, and the parameters type.

7.1 Lists

A list is a sequential collection of values where each value has a location (index) from 0 to $n-1$. Lists are heterogeneous, meaning they can store any type of value.

7.2 Stack

A stack is a collection of objects that follows the Last In, First Out (LIFO) principle.

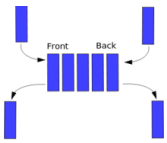


7.3 Queue

A queue is a collection of objects that follows the First In, First Out (FIFO) principle.

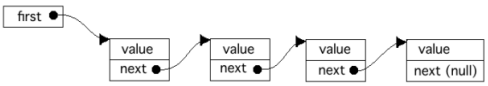
7.4 Deque

A deque is a double-ended queue, allowing elements to be added or removed from both ends.



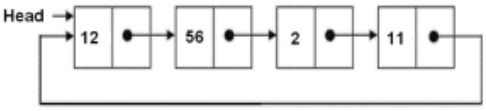
7.5 Linked Lists

A linked list is a data structure where elements are arranged in a linear order, maintained by pointers in each object. Each node contains a value and a pointer to the next element. The first element is the head, and the last is the tail.



7.5.1 Circularly Linked Lists

In a circularly linked list, the next pointer of the last element points to the head of the list.



7.5.2 Doubly Linked Lists

A doubly linked list has both a next and a previous pointer, with header and trailer sentinels for better symmetry.

7.6 Positional List

It is a data structure that allows us to perform arbitrary insertions and deletions or to refer to elements anywhere in a list. A positional list is an abstraction that provides the ability to identify the position of an element in a list.

Position	p	q	w
Element	e1	e2	e3

7.7 List VS Arrays

- Access complexity:
  - Arrays: $O(1)$ time access to an elements based on an index.
  - Lists: $O(n)$ in a linked list to do the traversal.
- Memory consumption for n elements:
  - Arrays: we may need to store $2n$ elements (dynamic resizing)
  - Lists: we store n elements and n references (singly) and $2n$ references (doubly).

Arrays	Linked Lists
An array is a collection of elements of a similar data type.	Linked List is an ordered collection of elements of the same type in which each element is connected to the next using pointers.
Array elements can be accessed randomly using the array index.	Random accessing is not possible in linked lists. The elements will have to be accessed sequentially.
Data elements are stored in contiguous locations in memory.	New elements can be stored anywhere and a reference is created for the new element using pointers.
Insertion and Deletion operations are costlier since the memory locations are consecutive and fixed.	Insertion and Deletion operations are fast and easy in a linked list.
Memory is allocated during the compile time (Static memory allocation).	Memory is allocated during the run-time (Dynamic memory allocation).
Size of the array must be specified at the time of array declaration/initialization.	Size of a Linked list grows/shrinks as and when new elements are inserted/deleted.

8 Trees

Formal Definition in Graph Theory: Let $V = v_1, ..., v_n$ be a set of nodes and the set E is a mapping of the set V in V, $E : V \rightarrow V$, thus $T(V, E) = T(V, VxV)$; E is defined as a set of couples v_i, v_j where $v_i, v_j \in V$ such that v_i is connected to v_j and thus v_i is the parent of v_j. If $T(V, E)$ is:

- **(i)** A connected graph of n vertices and $n - 1$ links.
- or **(ii)** A connected graph without a circuit.
- or**(iii)** A graph in which every part of vertices is connected with one and only one elementary path.

Then $T(V, E)$ is a tree.

8.1 Free trees

A graph G is a pair (V,E), where V is a finite set of vertices (nodes) and E is a binary relation on V (the set of edges). A free tree is a connected, acyclic, undirected graph. If a graph is acyclic but possibly disconnected, it is a forest.

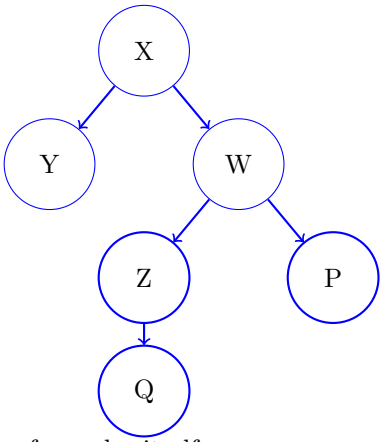
8.1.1 Properties of Free Tree

Let $G = (V, E)$ be an undirected graph. The following statements are equivalent:

1. G is a free tree.
2. Any two vertices in G are connected by a unique simple path.
3. G is connected, but if any edge is removed from E, the resulting graph is disconnected
4. G is connected and $|E| = |V| - 1$
5. G is acyclic and $|E| = |V| - 1$
6. G is acyclic, but if any edge is added to E, the resulting graph contains a cycle

8.2 Rooted Tree

A rooted tree is a free tree where one of the nodes is called root.



- Descendants(x)=y, w, z, p, q
- Ancestors(x)=
- Ancestor(q)=z, w, x
- Parent(x)=
- Children(q)=
- Children(w)=z, w

The subtree rooted at x is the tree induced by the descendants of x and x itself.
The number of children of a node x is defined as the degree of x.
An ordered tree is a rooted tree in which the children of each node are ordered; so, if x has 3 children, say y, w, z we can say that y is the first child, w is the second child ...

8.3 General Tree ADT

- p.element() → return the element stored at position p.
- T.root() → return the position of the root of the tree or none if it's empty.
- T.is_root(p) → return true if the node stored at position p is the root.

- T.parent(p) → return the position of the parent of the node stored in p.
- T.num_children(p) → return the number of children of p.
- T.children(p) → generate an iterator of the children of position p.
- T.is_leaf(p) → return true if p is a leaf.
- len(T) → return the number of nodes (positions) in T.
- T.is_empty() → return true if the tree is empty.

8.4 Depth of a Tree

```
depth(T,p)
Input: p belongs to T
Output: depth of p in T
d = 0
while (p = T.parent(p) is not none) do
    d = d + 1
return d
```

- Return the depth, the numbers from the root to the last element with a parent (leaf).
- **Running time:** $\Theta(n)$

8.5 Height of a Tree

```
height(T)
Input: T (the tree)
Output: height of T
h = 0
for each v belonging to T.positions() do
    if (T.isLeaf(v)) then
        h = max(h, depth(T,v))
return h
```

```
height2(T,p)
Input: p belongs to T (position of the
     node)
Output: height of p in T
if(T.isLeaf(p)) then
    return 0
else
h = 0
    for each w belongs to T.children(p
        ) do
        h = max(h, height2(T,w))
    return h+1
```

- The height of a tree is equal to the maximum of the depths of it's leaves.
- *T.position*() can be implemented in $O(n)$.
- **Running time:** $\Theta(n^2)$

8.6 Tree Traversals

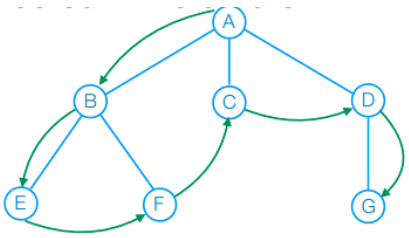
Tree traversal is a systematic method to visit the nodes of a tree executing an operation in each node.

8.6.1 Pre-order

Pre-order: you visit the parent and then the sub-trees rooted in the children

```
preorder(T,p)
    visit(p)
    for each c in T.children(p) do
        preorder(T,c)
Pre-order: A, B, E, F, C, D, G
```

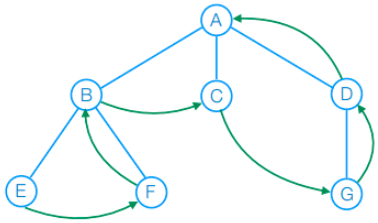
Running time: $O(n)$



8.6.2 Post-order

Post-order: you recursively visit the sub-trees rooted in the children and then the parent.

```
postorder(T,p)
    for each c in T.children(p) do
        postorder(T,c)
    visit(p)
Post-order: E, F, B, C, G, D, A
```

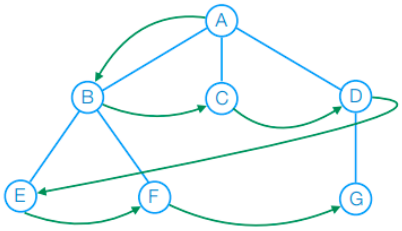


Running time: $O(n)$

8.6.3 Breadth First Traversal

You visit all the nodes at depth d before visiting all the nodes at depth d+1.

```
BFT(T,p)
    Q.enqueue(T.root())
    while !Q.isEmpty() do
        p = Q.dequeue()
        visit(p)
        for each c in T.children(p) do
            Q.enqueue(c)
BFT: A,B,C,D,E,F,G
```



Running time: $O(n)$

8.7 Binary Trees

A binary tree is an ordered tree such that:

- Every node has at most 2 children.
- Every child is labeled as **right node** or **left node**.
- The left child comes before the right child in the order of children node.

A binary tree is called **proper (or full)** if every node has zero or two children.
Let T be a proper binary tree with n nodes (where n_E are external nodes and n_I are internal nodes), then: $n_E = n_I + 1$

8.8 Binary Search Tree

A binary search tree T is a binary tree where if y is a node in the left subtree of x then $y.key \leqslant x.key$ and y is a key in the right subtree of x then $y.key \geqslant x.key$.

- $x.key$ is the same as $x.element()$.
- $x.left$ ($x.right$) returns the position of the left(right) child of x.

Is a rooted binary tree data structure whose internals nodes each store a key greater than all the keys in the node's left subtree and less than those in its right subtee. The core operations are:

- **Minimum:** get the minimum element is a search binary tree
- **Maximum:** get the maximum element is a search binary tree
- **Successor:** Given a number, find the successor in the sorted order
- **Predecessor:** Given a number, find the predecessor in the sorted order

8.8.1 Searching

We search for a node with a given key in a binary search tree.
Inputs: x is a given node where the search starts and k is the key to be searched in the tree.
Outputs: a pointer to a node with key k or NIL if the key is not in the tree.
Running time: $O(h)$ where h is the height, so $Olog(n)$.

```
ITERATIVE-TREE-SEARCH(x,k)
while x!=NIL and k!=x.key do
    if k<x.key
        x = x.left
    else
        x = x.right
return x
```

8.8.2 Minimum and Maximum

We search for the node with the maximum/minimum value.
Inputs: x is a given node where the search starts.
Outputs: a pointer to a node with maximum/minimum key.
Running time: $O(h)$ where h is the height, so $Olog(n)$.

```
TREE-MINIMUM(x)
while x.left!=NIL do
    x = x.left
return x

TREE-MAXIMUM(x)
while x.right!=NIL do
    x = x.right
return x
```

8.8.3 Successor

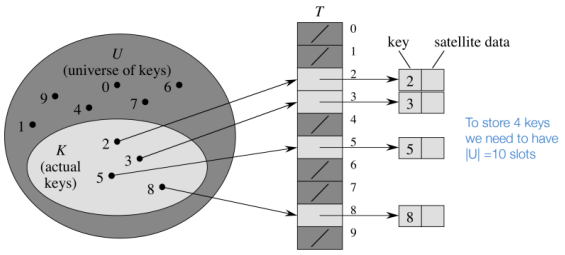
Given a BST T and x,y inT, then y is a **successor** of x if $y.key > x.key$ and there not exist a z in T such that $y.key > z.key > x.key$.
Property: Consider a BST T whose keys are distinct. It the right subtree of a node x in T is empty and x has a successor y, then y is the lowest ancestor of x whose left child is also an ancestor or self of x.
Inputs: x is a given node where the search starts.
Outputs: a pointer to a node with the successor key if exists, NILL otherwise.
Running time: $O(h)$ where h is the height, so $Olog(n)$.

```
TREE-SUCCESSOR(x)
if x.right!=NIL
    return TREE-MINIMUM(x.
        right)
else
    // go up the tree from x
        until we encounter a
    //node that is the left
        child of its parent
    y = T.parent(x)
    while y!=NIL and x==y.
        right
        x=y
        y = T.parent(y)
    return y
```

9 Hash Tables

9.1 Direct-address table

Direct accessing is a simple technique that works well when the universe U of possible keys is relatively small.
Given a universe $U = 0, 1, \ldots, m-1$ a direct-address table is an array $T[0, \ldots, m-1]$ where $T[i] = i \in U$.



You have **time efficiency**, because all the operations are done in $O(1)$, but the space is inefficient if $|U| >> |K|$.

```
Direct-Table-Search(T, k)
return T[k]

Direct-Table-Insert(T, x)
return T[x.key] = x

Direct-Table-Delete(T, x)
return T[x.key] = NIL
```

9.2 Hash Table

Use a table of size proportional to $|K|-$ the hash table. However, we lose the direct-addressing ability.

- Hash function h: mapping from U to the slots of a hast table $T[0, \ldots, m-1]$.
- $h : U \to 0, 1, \ldots, m-1$.
- With hast tables, key k hashes to slot $T[h[k]]$, where $h[k]$ is the hast value of k.

A hash table is an array of some fixed size, usually a prime number.

key space = integers

TableSize = 6

`h(K) = K mod 6`

Insert 7 7 mod 6 = 1

Insert 18 18 mod 6 = 0

Insert 41 41 mod 6 = 5

Insert 34 34 mod 6 = 4

Insert 10 10 mod 6 = 4

0	18	
1	7	
2		
3		
4	34 10	collision
5	41	

A hash function should: be simple and fast to compute, void collisions and have keys distributed evenly among cells.
Given that $|U| >> |K|$, collisions cannot be avoided, but can be handled.

9.3 Avoiding Collisions: Separate Chaining

- **INSERT** into the hash table requires $O(1)$ if we are sure that the element is not already present in the hash table otherwise it takes $O(1 + SEARCH)$.
- **SEARCH** worst-case running time is proportional to the length of the list associated with the hash.
- **DELETE** is $O(1 + SEARCH)$.

How long does **SEARCH** takes?

9.3.1 Loading Factor

Given a hash table T with m slots that store n elements, the load factor $\alpha = n/m$.
The load factor is the average number of elements stored in a hash table where $\alpha \in [0, 1]$. The worst case search time is $\theta(n)$. The average case for searching depends on how well the hash function distributes the n keys in the m slots (on average).

9.4 Simple Uniform Hashing

We assume that any given element is equally likely to hash into any of the m slots. For $j = 0, 1, \ldots\ m-1$, the list $T[j]$ is long n_j, so that $n = n_0 + n_1 + \cdots + n_{m-1}$. The expected value of n_j is $E[n_j] = a/n$.
The average running time for a successful search in a hash table solving collisions with chaining and adopting simple uniform hashing is $\theta(1 + \alpha)$.
If m is proportional to n, then $n = O(m)$ and $\alpha = n/m = O(m)/m = O(1)$.

9.5 Good Hash Function

We´d need to know the distribution of the keys. Most hash functions assume that the universe of keys is set of natural numbers.

9.5.1 Division Method

- $h(k) = kmodm$
- Choose m :
  - Prime not too close to a power of 2
  - $m = 2p$ means that $h(k)$ is just the p lowest-order bits of k (unless we know that the lower order bits are all equally likely, we´d better consider all the bits of the key).

We also have other methods like:

- **Multiplication Method:** $h(k) = \lfloor m(kAmod1) \rfloor$, $0 \leqslant A \leqslant 1$. A is $(\sqrt{5} - 1)/2 = 0.61803\ldots$
- **Squaring Method:** Square the key and then truncate.

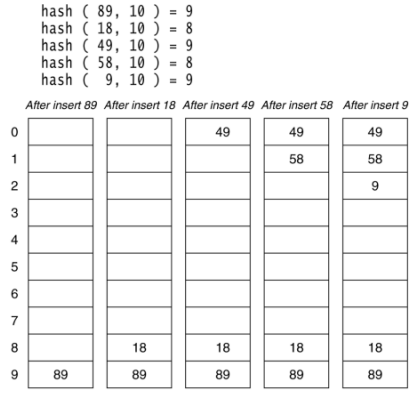
9.6 Open Addressing

All the data go inside the table and if a collision occurs, alternative cells are tried until an empty cell is found.
Cells $h_0(x), h_1(x)\ldots$ are tried in succession where $h_i(x) = (hash(x) + f(i))modTableSize$, with $f(0) = 0$.
The function f is the collision resolution strategy, there are three common collision resolution strategies:

9.6.1 Linear Probing

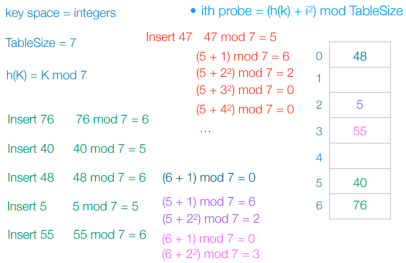
- Probe sequence:
  - 0^{th}probe= $h(k) = kmodTableSize$
  - 1^{st}probe= $(h(k) + 1) = kmodTableSize$
  - 2^{nd}probe= $(h(k) + 2) = kmodTableSize$
  - $\ldots$
  - i^{th}probe= $(h(k) + i) = kmodTableSize$

Add a function of i to the original hash value to resolve the collision. Aby key that hashes into cluster 1) will will require several attempts to resolve collision and 2) will then add to the cluster.



9.6.2 Quadratic Probing

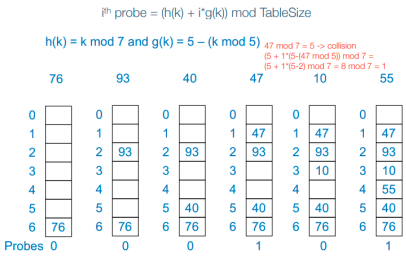
- Probe sequence with $f(i) = i^2$:
  - 0^{th}probe= $h(k) = kmodTableSize$
  - 1^{st}probe= $(h(k) + 1) = kmodTableSize$
  - 2^{nd}probe= $(h(k) + 4) = kmodTableSize$
  - ...
  - i^{th}probe= $(h(k) + i^2) = kmodTableSize$



Less likely to encounter primary clustering, but could run into secondary clustering. Although keys that hash to the same initial location still use the same sequence of probes. The hash table is twice as big as the number of the elements expected ($\alpha = 1/2$).// For any $\alpha < 1/2$ quadratic probing will find an empty slot, for bigger α quadratic probing may find a slot.

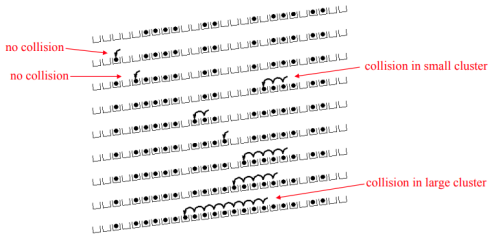
9.6.3 Double Hashing

- Probe sequence with $f(i) = i * g(k)$:
  - 0^{th}probe= $h(k) = kmodTableSize$
  - 1^{st}probe= $(h(k) + g(k)) = kmodTableSize$
  - 2^{nd}probe= $(h(k) + 2g(k)) = kmodTableSize$
  - ...
  - i^{th}probe= $(h(k) + i * g(k)) = kmodTableSize$



9.7 Primary and Secondary Clustering

It works pretty well for an empty table and gets worse as the table fills up. If a bunch of elements hash to the same spot, they clash one with the other. But, worse, if a bunch of elements hash to the same area of the table, they keep clashing.



9.8 Rehashing

When all tables are too full, create a bigger table and hash all the items from the original table into the new table. You re-hash when:

- Half full $\alpha = 0.5$
- When an insertion fails
- Other thresholds

The cost of rehashing is linear $O(n)$ but not good for real-time safety critical applications.

9.9 Python and Hash

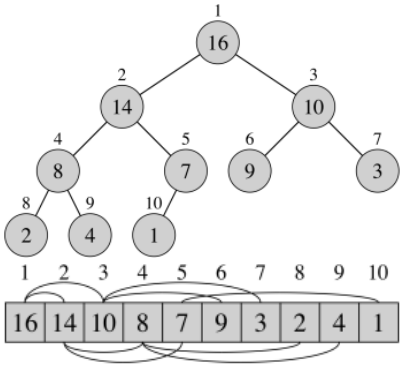
In Python dictionaries are used as hash tables and use random probing, where the next slot is picked in a pseudo random order.

10 Heap

The heap data structure is an array that can be represented as a complete binary tree, filled up at all levels with possibly an exception at the lowest level which is filled from the left.
An array A representing a heap is an object with two attributes: the $length[A]$ of the array and the $heap-size[A]$ which is the number of elements in the heap stored within the array ($heap-size \leqslant lenght$).

The heap has some **properties**:

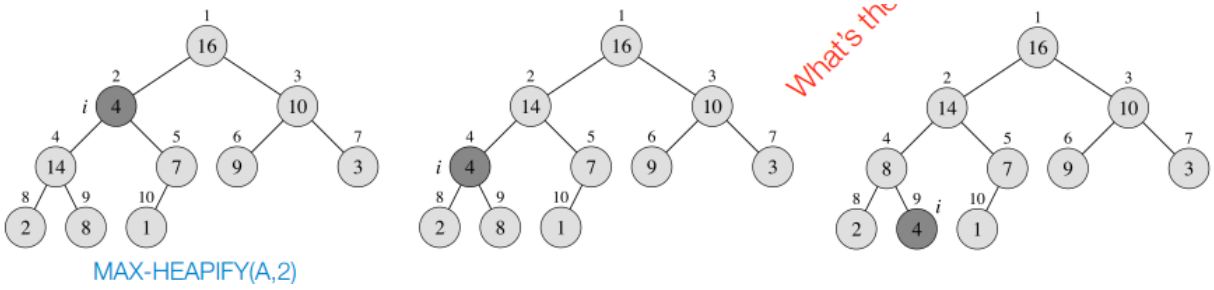
- The **Root** is always stored at $A[1]$.
- Given an element with index i
  - **Parent** is stored at $\lfloor i/2 \rfloor$.
  - **Left child** is stored at $2i$.
  - **Right child** is stored at $2i + 1$.
- Max-Heap property: $A[parent(i)] \geqslant A[i]$.
- Min-Heap property: $A[parent(i)] \leqslant A[i]$



10.1 Max-Heapify

Max-Heapify checks if the element at index i respect the max-heap property, if not it "floats down" the element in $A[i]$ until the property is required.
Running time: $\theta(log(n))$
You have the same algorithm also for Min-Heap.

```
Input: A is an array and an index i
MAX-HEAPIFY(A,i)
l = left(i)
r = right(i)
if l<=heap-size[A] and A[l]>A[i] then
    largest = l
else largest = i
if r,=heap-size[A] and A[r]>A[largest] then
    largest = r
if largest!=i then
    exchange A[i] with A[largest]
    MAX-HEAPIFY(A,largest)
```



10.2 Build-Max-Heap

With an array representation for storing an n-element heap, the leaves are the nodes indexed by $\lfloor n/2 \rfloor + 1, \lfloor n/2 \rfloor + 2, \ldots, n$
Running time: $On(n)$

```
Input: A is an array
BUILD-MAX-HEAP(A)
heap-size[A]=length[A]
for i=floor(length[A]/2)
    downto 1 do
    MAX-HEAPIFY(A,i)
```

10.3 Priority Queues

Heaps are used to implement efficient priority queues: max-priority queues and min-priority queues. A **priority queue** is a data structure to maintain a set S of elements, each with an associated key. It supports the following operations:

- **INSERT(S,x):** $S = SUx$. In list based $\rightarrow O(1)$, in heap $O(logn)$.
- **MAXIMUM(S):** return the element with the max key in the queue. In list based $\rightarrow O(n)$, in heap $O(1)$.
- **EXTRACT-MAX:** return and remove the element with the max key in the queue. In list based $\rightarrow O(n)$, in heap $O(logn)$.
- **INCREASE-KEY(S, x, k):** increase the value of x's key to k (k is assumed to be larger than the current x's key.

Max Priority Queues are used for instance for jobs scheduling in a computer where the key is the importance of the job.
Min Priority Queues are used for instance for event-driven simulations where the key of an event is its execution time.

11 Dynamic Programming

Break up a problem intoreconstructreconstruct a series of overlapping subproblems, and build up solutions to larger and larger subproblems. Is a way to speed-up certain classes of inefficient recursive algorithms.

11.1 Elements of Dynamic Programming

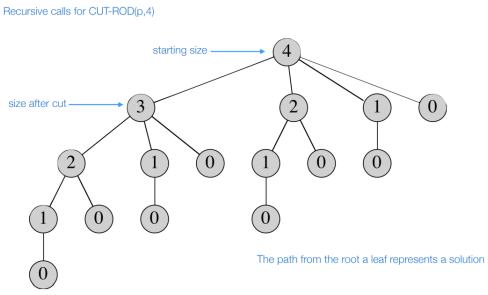
- **Simple Subproblems:** we should be able to break the main problem into smaller subproblems sharing the same structure.
- **Optimal Substructure of the Subproblems:** the optimal solution of a problem contains within the optimal solution of its subproblems.
- **Overlapping Subproblems:** There exists a place where the same subproblems are solved more than once.

11.2 Rod Cutting

Problem: Given a rod of length n inches and a table of prices, determine the maximum revenue obtainable by cutting up the road and selling the pieces. Rod cuts are an integer number of inches, cuts are free and you have a price table for rods:

- $r_n = max(p_n, r_1 + r_{n-1}, r_2 + r_{n-2} + \cdots + r_{n-1} + r_1)$ so $r_n = max(p_i + r_{n-i}) for 1 \leqslant i \leqslant n$.
- p_n = no cuts and selling the rod as is.
- The other $n - 1$ arguments to max correspond to the maximum revenue obtained by making an initial cut of the rod into two pieces of size i and $n - i$, for each $i = 1, 2, \ldots, n - 1$. Then optimally cutting up those pieces further, obtaining revenues r_i and r_{n-i} from those two pieces.
- Once we make the first cut, we may consider the two pieces as independent instances of the rod-cutting subproblem.
- The overall optimal solution incorporates optimal solutions to the two related subproblems, maximizing revenue from each of those two pieces.

```
CUT-ROD(p, n)
if n == 0
    return 0  ← no revenue is possible, and so CUT-ROD returns 0
q = −∞
for i = 1 to n
    q = max(q, p[i] + CUT-ROD(p, n − i))← Recursive call on the second piece
return q
```



The **Running Time** is exponential, $T(n) = 2^n$. The idea is to organise the algorithm in order to solve the subproblems only once, store the solution in an appropriate data structure and retrieve the solution every time we encounter an already solved subproblem (this is a **time-memory trade off**.
Dynamic Programming runs in polynomial time when the size of the input is polynomial, the number of the distinct subproblems is polynomial and each subproblem can be solved in polynomial time.

11.3 Memoization

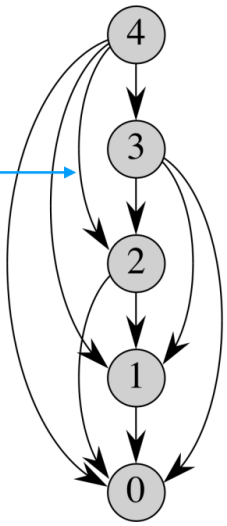
There are two approaches:

- **Top-Down Memoization:** we solve the problem in the usual way but storing the solutions of the sub-problems in the way.
- **Bottom-Up Memoization:** defines the sub-problems, order them by size in increasing order, solve them and go on solving the rest by using the memorised solutions.

11.3.1 Top-Down Memoization

```
MEMOIZED-CUT-ROD(p, n)
  let r[0..n] be a new array
  for i = 0 to n
      r[i] = −∞
  return MEMOIZED-CUT-ROD-AUX(p, n, r)
```

We need the solution of "3" when solving "4"




```
MEMOIZED-CUT-ROD-AUX(p, n, r)
  if r[n] ≥ 0
      return r[n]          ← check for the stored solution
  if n == 0
      q = 0
  else q = −∞
      for i = 1 to n
          q = max(q, p[i] + MEMOIZED-CUT-ROD-AUX(p, n − i, r))
  r[n] = q                 ← store the solution for n
  return q
```

11.3.2 Bottom-Up Memoization

```
BOTTOM-UP-CUT-ROD(p, n)
  let r[0..n] be a new array
  r[0] = 0
  for j = 1 to n
      q = −∞
      for i = 1 to j
          q = max(q, p[i] + r[j − i])
      r[j] = q
  return r[n]
```

- It uses the natural ordering of the subproblems: a problem of size i is "smaller" than a subproblem of size j if $i < j$.
- The procedure solves subproblems of sizes $j = 0, 1, \ldots, n$, in that order.

The running time of both problems is $\Theta(n^2)$. Each subproblem is solved just once, so the running time is the sum of the rimes needed to solve each subproblem.
Then you need to extend this to the optimal solution:

```
EXTENDED-BOTTOM-UP-CUT-ROD(p, n)
  let r[0..n] and s[1..n] be new arrays
  r[0] = 0
  for j = 1 to n
      q = −∞
      for i = 1 to j
          if q < p[i] + r[j − i]
              q = p[i] + r[j − i]
              s[j] = i     ← stores the optimal size i of the first piece to cut off when solving a subproblem of size j.
      r[j] = q
  return r and s
```

```
PRINT-CUT-ROD-SOLUTION(p, n)
  (r, s) = EXTENDED-BOTTOM-UP-CUT-ROD(p, n)
  while n > 0
      print s[n]
      n = n − s[n]
```

11.4 Longest Common Subsequence

Given 2 sequences x and y determine a longest subsequence.
Example: x: ABCBDAB y: BDCABA → BDAB, BCAB, BCBA

11.4.1 Brute Force Algorithm

Check every subsequence of $x[1 \dots m]$ to see if it is also a subsequence of $y[1 \dots n]$. **Analysis:**

- Given a subsequence of x, how long does it takes to check if it is a subsequence of y? $O(n)$.
- How many subsequences are there in x? 2^m.
- What is the brute force complexity? $(n * 2^m) \rightarrow m >> n \rightarrow O(2^m)$.

11.4.2 Simplification

1. Look at the length of LCS(x,y) (this length is unique).
2. Extend the algorithm to find our LCS(x,y) ($|S| \rightarrow$ length of S).

Strategy: consider only prefixes of x and y and we are going to show how we can express the length of the LCS of prefixes in terms of each others.
Define: $c[i,j] = |LCS(x[1 \dots i], y[1 \dots j]|$ and calculate $C_{ij}, \forall i,j \in n,m$.
How do we solve the problem of LCS(x,y) if we have $C_{ij}, \forall i,j$? We want to express $c[m,n] = |LCS(x,y)|$.

Theorem: $C_{ij} = \begin{cases} 0 & \text{if } i = 0 \text{ or } j = 0 \\ c[i-1,j-1]+1 & \text{if } x[i] = y[j] \\ max\{c[i,j-1], c[i-1,j]\} & \text{otherwise} \end{cases}$

This i s a recursive algorithm for LCS that is not a brute force algorithm.
The complexity is worst than the brute force algorithm ($O2^{n+n}$), so we need to improve.

reconstruct

```
LCS(x,y,i,j)
if x[i]==y[j] then
    c[i,j]=LCS(x,y,i-1,j-1)+1
else
    c[i,j]=MAX{LCS(x,y,i-1,j),LCS(x,y,i,j-1)}
return c[i,j]
```

11.4.3 Dynamic Programming Algorithm

```
LCS-LENGTH(x,y,m,n)
    let b[1...m,1...n] and c[0...m,0...n]
    for i=1 to m
        c[i,0]=0
    for j=1 to n
        c[0,j]=0
    for i=1 to m
        for j=1 to n
            if x_i==y_j then
                c[i,j]=c[i-1,j-1]+1
                b[i,j]="diagonal␣arrow"
            else if c[i-1,j]>=c[i,j-1] then
                c[i,j]=c[i-1,j]
                b[i,j]="up␣arrow"
            else
                c[i,j]=c[i,j-1]
                b[i,j]="left␣arrow"
    return c and b
```

```
PRINT-LCS(b,x,y,i,j)
    if i==0 or j==0
        return m
    if b[i,j]=='diagonal␣arrow' then
        PRINT-LCS(b,x,i-1,j-1)
        print Xi
    elseif b[i,j]=='up␣arrow' then
        PRINT-LCS(b,x,i-1,j)
    else
        PRINT-LCS(b,x,i,j-1)
```

11.5 Text Justification

Given a sequence of words, and a limit on the number of chars that can be put in one line (line width), put the breaks in the given sequence such that the lines are printed neatly.
Input: A given array of words $l = w[0 \dots n-1]$ and $|l| = n$.
Scoring rule: suppose we are considering the words from i to j $\rightarrow w[i]$ *to* $w[j], w[i \dots j]$, then we define the badness of a justification as:

$$badness(i,j) = \begin{cases} +\infty & \text{if } j+1-i > \text{ page-width} \\ (page - width - (j+1-i))^2 & \text{otherwise} \end{cases}$$

Goal: split words into lines $l_1 = w[0 \dots i_1 - 1], l_2 = w[0 \dots i_2 - 1], \dots$ to minimize $\sum_i badness(l_i)$.

11.5.1 Subproblem Structure

1. **Subprreconstructoblem:** $c[i] = min\ badness(i, n-1)$
   - N *subproblems* $= \Theta(n)$, *where* $n = N$ *words*
2. **Guessing:** where to end the first line in the optimal justification of words
   (a) $w[i \dots n-1] \rightarrow N$ *choices* $= n - i + 1 \rightarrow O(n)$.
3. **Recurrence:** $c[i] = min(badness(i,j) + c[j+1] \forall j \in [i, n-1]$.

Example of a greedy approach:
Words: diamonds are girls best friends, page-width = 12

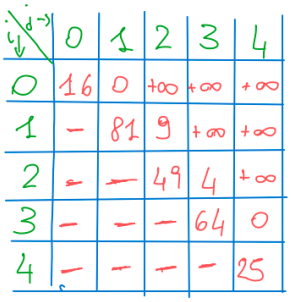
i		length
0	diamonds	8
1	are	3
2	girls	5
reconstruct 3	best	4
4	friends	7

Greedy algorithm

words	badness
diamonds are	0
girls best	2^2
friends	5^2
	29

DP algorithm

words	badness
diamonds	4^2
are girls	3^2
best friends	0
	25



We need to determine 2 arrays, one for the cost and one to reconstruct the optimal solution.

11.5.2 Dynamic Pseudocode

Input: an array of words and p_w.
Output: minCost and the index.
Let badness[0 ... len(W)-1, minCost0 ... len(W)-1] be an empty matrix

```
for i=0 to len(W)-1 do
    badness[i,i] = page_width - len(W[i])
    for j=i+1 to len(W)-1 do
        badness[i,j]=badness[i,j-1]-len(w[j])-1
for i=0 to len(W)-1 do
    for j=i to len(W)-1 do
        if badness[i,j]<0 then
            badness[i,j]=+inf
        else
            badness[i,j]=badness[i,j]^2
```

Find the minCost[0 ... len(w)-1] by using the given recurrence.
Let minCost[0 ... n-1] and index[0 ... n-1] be 2 empty arrays.

```
for i=n-1 to 0 do
    minCost[i]=badness[i,n-1]
    index[i]=n-1
    for j=n-1 to i+1 do
        if badness[i,j-1]!=inf do
            if minCost[i]>badness[i,j-1]+minCost[j] then
                minCost[i]=badness[i,j-1]+minCost[j]
                index[i]=j
return (minCosts, index)
```

```
Print_Justified_Text(w,page_width)
    (minCost,index)=Text_Justification(w,page_width)
    i=0
    do
        j=index[i]
        for k=1 to j-1 do
            if k != j-1 then
                print(w[k]+"␣")
            else
                print(w[k])
        print("\n")
        i=j
    while j<length(w)
```

12 Graphs

A graph $G = (V, E)$ has $V = set\ of\ vertices, E = set\ of\ edges$ and $|E| = O(|V|^2)$. A graph can be

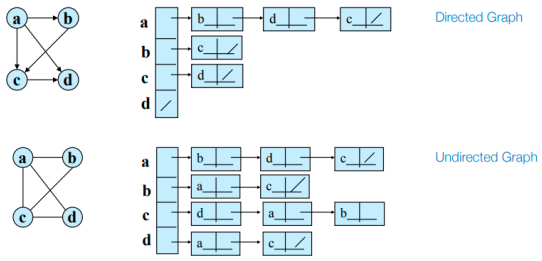
- **Undirected:** no self loops
- **Directed:** self loops allowed
- **Weighted:** $E \to R$
- **Dense:** $|E| \sim |V|^2$
- **Sparse:** $|E| << |V|^2$

If $(u, v) \in E$, then vertex v is adjacent to vertex u. An **Adjacency Relationship** is symmetric if G is undirected or not necessarily if is directed.
If G is connected there is a path between every pair of vertices, $|E| \geqslant |V| - 1$ and if $|E| = |V| - 1$, then G is a tree.

12.1 Graph Representation

12.1.1 Adjacency Lists

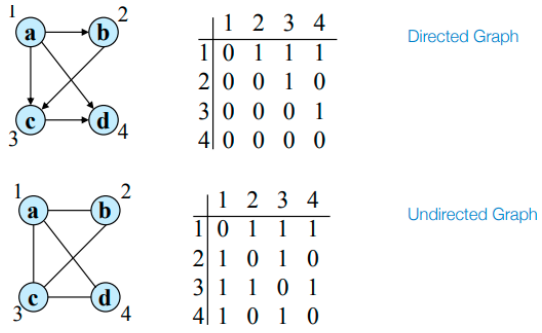
- Consists of an array Adj of $|V|$ lists, with one list per vertex.
- For $u \in V, Adj[u]$ consists of all vertices adjacent to u.
- **Storage requirements:**
  - directed $\Theta(|V| + |E|)$,
  - undirected $\Theta(|V| + |E|)$



Pros: space-efficient when a graph is sparse and can be modified to support many graph variants.
Cons: determining in an edge $(u, v) \in G$ is not efficient, have to search in u's adjacency list costs $\Theta(degree(u)) - time$ and $\Theta(V)$ in the worst case.

12.1.2 Adjacency Matrix

- $|V|x|V|$ matrix A.
- Number vertices from 1 to $|V|$ in some arbitrary manner.
- A is then given by
  $A[i, j] = a_{ij} = \begin{cases} 1\ if\ (i, j) \in E \\ 0 otherwise \end{cases}$



The **space** required is $\Theta(V^2)$ (not efficient for large graphs), the **time** to list all vertices adjacent to u is $\Theta(V)$ and to determine if $(u, v) \in E : \Theta(1)$. Can store weight instead of bits for weighted graph.

12.2 Nomenclature

- A path of length k from a vertex u to v in a graph $G = (V, E)$ is a sequence: $p = \langle u, u_1, u_2, \ldots, v \rangle$ where $(u, u_1), (u_1, u_2), \ldots$ belong to E. The length of the path is denoted as $|p| = k$.
- In a graph G, we say that a node v is **reachable** from u if there exists a path in G from u to v.

- A path is **simple** if all vertices in the path are distinct.
- An undirected graph G is **connected** if there exists a path between every pair of vertices.
- A graph G is **strongly connected** if every two vertices are reachable from each other.

12.3 Search a Graph

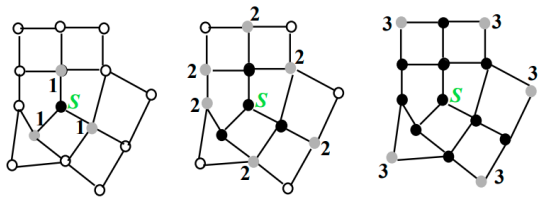
12.3.1 Breadth-First-Search (BFS)

Input: Graph $G = (V, E)$, either directed/indirected, and source vertex $s \in V$.
Output:

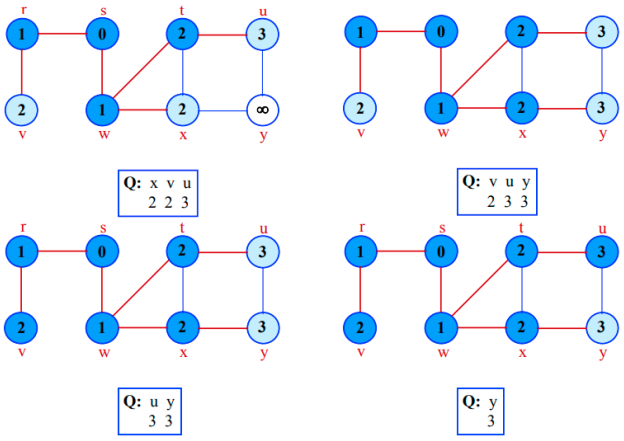
- $d[v]$ = distance (smallest number of edges, or shortest path) from s to v, for all $v \in V$. $d[v] = \infty$ if v is not reachable from s.
- $\pi[v] = u$ such that (u, v) is last edge on shortest path $s \to v$, so u is v predecessor.
- Builds a tree with root s that contains all reachable vertices.

A node can be:

- **White:** undiscovered.
- **Grey:** discovered but not finished.
- **Black:** finished.



```
BFS(G,s)
for each vertex u in V[G]-{s} do
color[u] = white
    d[u] = inf
    pi[u] = nil
color[s] = grey
d[s] = 0
pi[s] = nil
initialise the queue Q
Q.enqueue(s)
    while Q != empty do
        u = Q.dequeue()
        for each v in Adj[u] do
            if color[v] == white then
                color[v] = grey
                d[v] = d[u] + 1
                pi[v] = u
                Q.enqueue(v)
        color[u] = black
```



reconstructreconstructreconstruct **Complexity:** $O|V| + |E|$ with AdjMatrix and $\Theta(|V| + |E|$ with AdjList.

12.3.2 Breadth-First Tree

For a graph $G = (V, E)$ with source s, the **predecessor subgraph** of G is $G' = (V', E')$ where:

- $V' = \{v \in V : [v] \neq \text{NIL}\} \cup \{s\}$
- $E' = \{([v], v) \in E : v \in V' - \{s\}\}$

The predecessor subgraph G' is a **breadth-first tree** if:

- V' consists of the vertices reachable from s, and
- For all $v \in V'$, there exists a unique simple path from s to v in G' that is also the shortest path from s to v in G.

The edges in E' are called **tree edges** and the number of edges satisfies $|E'| = |V'| - 1$.

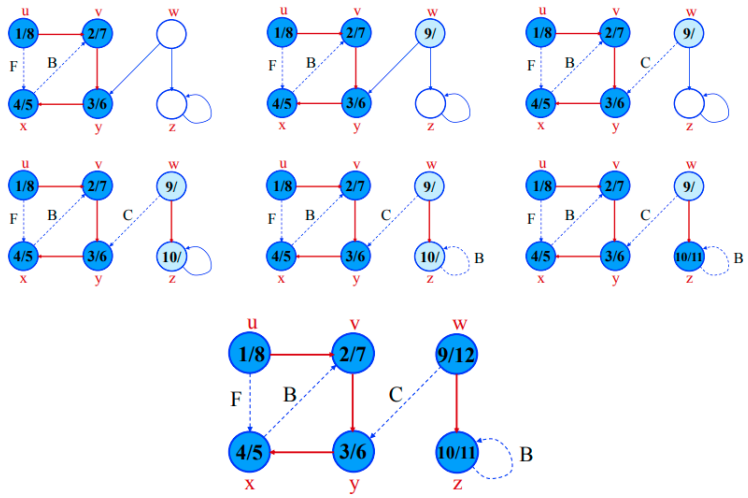
12.3.3 Depth-First-Search (DFS)

Input: $G = (V, E)$ direct of undirected, no source vertex given.
Output:

- 2 timestamps on each vertex, integers between 1 and $2|V|$.
- $d[v]$ = discovery time (v turns from white to gray.
- $f[v]$ = finishing time (v turns from gray to black)
- $\pi[v]$ = predecessor of $v = u$, such that v was discovered during the scan of u adj list.

```
DFS(G)
for each vertex u in V[G]do
    color[u] = white
    pi[u] = nil
time = 0
for each vertex u in V[G] do
    if color[u]==white then
        DFS-Visit(u)

DFS-Visit(u)
color[u]= grey
time = time+1
d[u] = time
for each v in Adj[u] do
    if color[v]==white then
        pi[v] = u
        DFS-Visit(v)
color[u] = black
f[u] = time
time = time + 1
```



Complexity: $O|V| + |E|$ with AdjMatrix and $\Theta(|V| + |E|$ with AdjList.

12.3.4 Classification of Edges

- **Tree Edge:** in the depth-first forest.
- **Back Edge:** (u, v) where u is a descendant of v (in the depth-first tree).
- **Forward Edge:** (u, v) where u is a descendant of u, but not a tree edge.
- **Cross Edge:** any other edge. Can go between vertices in same depth-first tree or in different depth-first trees.

12.3.5 DFS Trees

Predecessor subgraph defined slightly different from that of BFS. The predecessor subgraph of DFS is $G\pi = (V\pi, E\pi)$ where $E\pi = (\pi[v], v) : v \in V and \pi[v] \neq NIL$.
The predecessor subgraph $G\pi$ forms a depth-first forest composed of several deoth-first trees.
The edges in $E\pi$ are called tree edges.

12.4 Single Source Shortest Path

The shortest path algorithm calculates the shortest (weighted) path between a pair of nodes. It's useful for user interactions and dynamic workflows because it works in real time.
Use shortest path to find optimal routes between a pair of nodes, based on either the number of hops or any weighted relationship value. It can provide real-time answers about degrees of separation, the shortest distance between points, or the least expensive route.

- Let $G(V, E)$ a weighted graph s.t. there exists $w : E \to R$.
- We define a direct path from v_1 to v_k as $v_1 \to v_k$ as $p = v_1 \to v_2 \to \ldots v_{k-1} \to v_k$.

- The weight of p is $w(p) = \sum_i W(v_i, v_{i+1}$ and can be positive, negative or zero.
- Shortest path from u to v is the $p_{u,v}$ with minimum weight.
- A simplification of this problem is to calculate the weight of the shortest path rather than the path itself $\delta(u, v) = min(w(p) : p \ from \ u \ to \ v)$.

12.4.1 Dijkstra Algorithm

- Operates by first finding the lowest-weight relationship from the start node to directly connected nodes.
- It keeps track of those weights and moves to the closest node.
- It then performs the same calculation, but now as a cumulative total from the start node.
- The algorithm continues to do this, evaluating a wave of cumulative weights and always choosing the lowest weighted cumulative path to advance along, until it reaches the destination node.
- Q is a Min Priority Queue.

```
DIJKSTRA(G, w, s)
  INIT-SINGLE-SOURCE(G, s)
  S = empty
  for each vertex u in G.V
      INSERT(Q, u)
  while Q != empty
      u = EXTRACT-MIN(Q)
      S = S U \{u}
      for each vertex v in G.Adj[u]
          RELAX(u, v, w)
          if v.d changed
              DECREASE-KEY(Q, v, v.d)
```

```
INIT-SINGLE-SOURCE(G, s)
  for each v \in G.V
      v.d = inf
      v.pi = NILL
  s.d = 0

RELAX(u, v, w)
  if v.d > u.d + w(u, vreconstruct)
      v.d = u.d + w(u, v)
      v.pi = u
```

- **Triangle Inequality:** for any edge $(u, v) \in E$, we have $\delta(s, v) \leqslant \delta(s, u) + w(u, v)$.
- **Upper-Bound Property:** we always have $v.d > \delta(s, v)$ for all vertices $v \in V$, and once v.d achieves the value $\delta(s, v)$, it never changes.
- **No-Path Property:** If there is no path from s and v, then we always have $v.d = \delta(s, v) = \infty$.
- **Convergence Property:** If $s \to u \to v$ is a shortest path in G for some $u, v \in V$ and if $u.d = \delta$ at any time prior to relaxing edge (u, v), then $v.d = \delta(s, v)$ at all times afterwards.

12.5 Minimum Spanning Tree

The **Minimum Spanning Tree** algorithm starts from a given node and finds all its reachable nodes and set of relationships that connect the nodes together with the minimum possible weight.
Prim's algorithm rather than minimizing the total length of a path ending at each relationship, it minimizes the length of each relationship individually, so returns the global minimum (negative weights are allowed).

12.5.1 Prim's MST

- It begins with a tree containing only one node.
- The relationship with smallest weight coming from that node is selected and added to the tree.
- This process is repeated, always choosing the minimal-weight relationship that joins any node not already in the tree.
- When there are no mode nodes to add, the tree is a minimum spanning tree.

```
PRIM(G,w,r)
   Q=empty
   for each u in G.V
      u.key=inf
      u.pi=NIL
      INSERT(Q,u)
   DECREASE-KEY(Q,r,0)
   while Q != empty
      u=EXTRACT-MIN(Q)
      for each v in G.Adj[u]
         if v in Q and w(u,v) < v.key
             v.pi=u
             DECREASE-KEY(Q,v,w(u,v))
```

12.5.2 When to use MST

Use the MST when you need the best route to visit all nodes. Is useful when you must visit all nodes in a single walk.
It's also employed to approximate some problems with unknown compute times, such as the Travelling Salesman Problem.

12.6 Centrality Algorithms

Centrality algorithms are used to understand the roles of particular nodes in a graph and their impact on that network.
They're useful because they identify the most important nodes and help us understand group dynamics such as credibility, accessibility, the speed at which things spread, and bridges between groups.

12.6.1 Degree Centrality

Measures the number of relationships a node has. Is estimating a person's popularity by looking at their in-degree and using their out-degree to estimate gregariousness.

$$C_D(v) = \frac{\deg(v)}{n-1}$$

- $C_D(v)$: The **degree centrality** of node v, which measures the importance of v based on its connections.
- $\deg(v)$: The **degree** of node v, representing the number of edges connected to v.
- n: The **number of nodes** in the graph.

12.6.2 Closeness Centrality

Calculates which nodes have the shortest paths to all other nodes. Is finding the optimal location of new public services for maximum accessibility.
Is a way of detecting nodes that are able to spread information efficiently through a sub-graph. Nodes with a high closeness score have the shortest distances from all other nodes.

$$C_C(v) = \frac{n-1}{\sum_{u \in V \setminus \{v\}} d(v,u)}$$

- $C_C(v)$: The **closeness centrality** of node v, which measures how close v is to all other nodes in the graph.reconstruct
- n: The **number of nodes** in the graph.
- $d(v,u)$: The **shortest path distance** between node v and node u.
- V: The set of all nodes in the graph.

$$C_C^{\text{norm}}(v) = \frac{n-1}{\sum_{u \in V \setminus \{v\}} d(v,u)}$$

- $C_C^{\text{norm}}(v)$: The **normalized closeness centrality** of node v, which scales the closeness centrality to account for the number of nodes in the graph.
- n: The **number of nodes** in the graph.
- $d(v,u)$: The **shortest path distance** between node v and node u.
- V: The set of all nodes in the graph.
- The normalization ensures that the closeness centrality is bounded between 0 and 1 for all graphs.

$$C_C^{\text{WF}}(v) = \frac{n-1}{\sum_{u \in V \setminus \{v\}} d(v,u)}$$

- $C_C^{\text{WF}}(v)$: The **Wasserman and Faust closeness centrality** of node v, which measures how close v is to all other nodes in the graph, normalized by the size of the graph.
- n: The **number of nodes** in the graph.
- $d(v,u)$: The **shortest path distance** between node v and node u.
- V: The set of all nodes in the graph.

12.6.3 Harmonic Centrality

Harmonic Centrality solves the original problem with unconnected graphs.

$$C_H(v) = \sum_{u \in V \setminus \{v\}} \frac{1}{d(v,u)}$$

- $C_H(v)$: The **harmonic centrality** of node v, which measures how efficiently v can access other nodes in the graph.
- $d(v,u)$: The **shortest path distance** between node v and node u. If no path exists, $d(v,u)$ is considered infinite, and $1/d(v,u)$ is treated as 0.
- V: The set of all nodes in the graph.

12.6.4 Betweenness Centrality

Measures the number of shortest paths that pass through a node. Is improving drug targeting by finding the control genes for specific diseases.
Is a way of detecting the amount of influence a node has over the flow of information or resources in a graph. Is used to find nodes that serve as a bridge from one part of a graph to another.
A node in considered **pivotal** for two other nodes if it lies on every shortest path between those nodes. If you remove a pivotal node, the new shortest path for the original node pairs will be longer or no more costly.

$$C_B(v) = \sum_{s \neq v \neq t \in V} \frac{\sigma_{st}(v)}{\sigma_{st}}$$

- $C_B(v)$: The **betweenness centrality** of node v, which quantifies the importance of v in terms of the numberreconstruct of shortest paths passing through it.
- σ_{st}: The total number of shortest paths between nodes s and t.
- $\sigma_{st}(v)$: The number of those shortest paths that pass through node v.
- V: The set of all nodes in the graph.

12.6.5 Page Rank

Estimates a current node's importance from its linked neighbours and their neighbours. Is finding the most influential features for extraction in machine learning.
Is the best known of centrality algorithms and it measures the transitive influence of nodes. It considers the influence of a node's neighbours, and their neighbours.

$$PR(u) = (1-d) + d\left(\frac{PR(T1)}{C(T1)} + \cdots + \frac{PR(Tn)}{C(Tn)}\right)$$

- $PR(u)$: The **PageRank** score of node u.
- d: The **damping factor**, typically set to 0.85. Defines the probability that the next click will be through a link.
- $PR(Ti)$: The PageRank score of node Ti, where Ti is a node pointing to u.
- $C(Ti)$: The **out-degree** of node Ti, i.e., the number of outgoing edges from Ti.

A difficulty is created by nodes that point only to each other in a group. Circular references cause an increase in their ranks as the surfer bounces back and forth among the noes. There are two strategies to avoid rank sinks:

- When a node is reached that has no outgoing relationships, PageRank assumes outgoing relationships to all nodes. Traversing these invisible links is sometimes called **teleportation**.
- The damping factor provides another opportunity to avoid sinks by introducing a probability for direct link versus random node visitation. When you set d to 0.85, a completely random node is visited 15% of the time.

12.7 All Pairs Shortest Path

The APSP algorithm calculates the shortest (weighted) path between all pairs of nodes.
It optimizes operations by keeping track of the distances calculated so far and running on nodes in parallel. Is commonly used for understanding alternative routing when the shortest route is blocked or becomes suboptimal.

12.7.1 Floyd-Warshall Algorithm

Input: $G(V, E)$
Complexity: $\Theta(V^3)$
We can have weights on edges but no negative weights cycles.
Given a path $p =< v_1, v_2, \ldots, v_a > \rightarrow \{v_2, v_2, \ldots, v_{a-1}\}$
Lemma 1: Given a weighted, directed graph $G = (V, E)$ with weights function $w : E \rightarrow R$, let $p =< v_0, v_2, \ldots, v_k >$ be the shortest path from v_o to v_k and for any $i, j | o \leqslant i \leqslant j \leqslant k$ let $p =< v_i, v_{i+1}, \ldots, v_j >$ be the subpath of p from v_i to v_j. Then, $p_{i,j}$ is a shortestpath from v_i to v_j.
Recursive Solution
$d^k_{i,j} \rightarrow$ weight of the SP from i to j with intermediates in $1, 2, \ldots, k$
$d^0_{i,j} = p = i \rightarrow j$ no intermediate
$d^0_{i,j} = w_{ij}$
$d^k_{i,j} = \begin{cases} w_{ij} if k = 0 \\ min(d^{k-1}_{i,j}, d^{k-1}_{ik} + d^{k-1}_{kj}, k \geqslant 1 \end{cases}$
$D^{(n)} = (d^{(n)}_{i,j}$
Recursive Solution Input: $\begin{cases} 0 \quad if \quad k = 0 \\ w[i,j] \quad if \quad i \neq j \quad and \quad (i,j) \in E \\ +\infty \quad if \quad i \neq j \quad and \quad (i,j) \notin E \end{cases}$

```
Floyd-Warshall(W)
    n=rows[W]
    D^(0)=W
    for k=1 to n do
        for i=1 to n do
            for j=1 to n do
                d_ij^(k)=min(d_{ij}^{k-1},d_{ik}^{k-1}+d_{ki}^{k-1})
    return D^{(n)}
```