COSC262: Algorithms
Part 1 (weeks 1 to 6)

Kourosh Neshatian – 2021

Contents

1 Introduction

Algorithms are all around us. Examples are:

- operating systems
- networks
- biometric algorithms
- barcode / QR code (matrix barcode) reader/scanner
- web search
- traffic lights, cars (ECUs), ...

1.1 Algorithms

An **algorithm** is a well-defined computational procedure that achieves a specified task. Normally, the procedure takes as its *input* an instance of a **computational problem**, and produces as its *output* a solution to the instance of the problem. The procedure needs to work for all instances of the problem.

A computational problem must be *general* and *well-specified*. It describes the input-output relationship: the set of possible inputs, their properties, and the desired output for each input.

Example 1.1. Formally define the sorting problem.

Problem: sorting

Input: a sequence of objects $a_1, a_2, \ldots, a_n$ which are totally ordered:

$$\forall i, j \in \{1, 2, \ldots, n\} : a_i \leq a_j \vee a_j \leq a_i$$

Output: a permutation π of $1, 2, \ldots, n$ such that

$$a_{\pi(1)} \leq a_{\pi(2)} \leq \ldots \leq a_{\pi(n)}$$

$\triangle$

Ways to describe algorithms:

- Structured English
- Pseudocode
- Program

1.2 Desired Properties

We are interested in algorithms that have the following properties (listed in their order of importance):

- Correct
- Scalable:
  - efficient in time (i.e. fast)
  - efficient in space (i.e. do not require too much memory)
  - efficient in other terms that matter (e.g. power consumption, core utilisation, anytime, etc)
- Others: general, simple, clever, etc.

1.3 Correctness of algorithms

An algorithm is said to be **correct** if for every input instance, it halts and produces the desired output.

The only way to ascertain the correctness of an algorithm is to provide a formal **proof**. A proof uses a set of axioms, assumptions, and a chain of deductive reasoning to establish the correctness of a claim.

Common proof techniques in Computer Science include *induction* and *proof by contradiction*.

Question 1.1. Can test cases be used to prove the correctness of an algorithm?

Critical thinking is important in designing algorithms. Before trying to prove the correctness of an algorithm it is worth trying to prove its incorrectness by finding a counterexample. A **counterexample** for an algorithm is an instance of a problem for which the algorithm does not produce the correct output. A counterexample is a proof of the incorrectness of the algorithm.

Refer to the textbook to see a Job Scheduling problem (Section 1.2), the Travelling Salesman Problem (TSP) (section 1.1), and a collection suggestions on how to find counterexamples (Section 1.3.3).

2 Analysis

The most important property of an algorithm is its *correctness*. Once we know an algorithm is correct, we need to know how efficient it is – that is, how much computational resources it needs to complete its task. Efficiency matters because it determines how scalable an algorithm is.

Among different types of resources that are required by an algorithm, we are usually interested in knowing

- how much *time* it would take to solve a problem (time complexity);
- how much *space* (memory) it requires to do so (space complexity).

Complexities are expressed in the form of a function of some property of the input (usually size).

2.1 The RAM Model of Computation

Talking about efficiency of algorithms without having a computational model in mind is not possible. Yet we do not want to be confined to a specific computing platform (hardware, operating system, programming language, etc). We need an abstract model that a) is close enough to common real computers; and b) is simple enough to reason about.

The *RAM model of computation* is an abstraction of real-world computers that is commonly used in analysis of algorithms. It assumes:

- Accessing any memory cell takes a constant time irrespective of its location and content (1 step)
- Basic operations such as arithmetic $(+, -)$, comparisons $(\leq, =)$ and assignments take a constant time (1 step)
- Loops take time proportional to the number of times they iterate.
- The time a subroutine call takes is equal to the time it takes to execute its body.

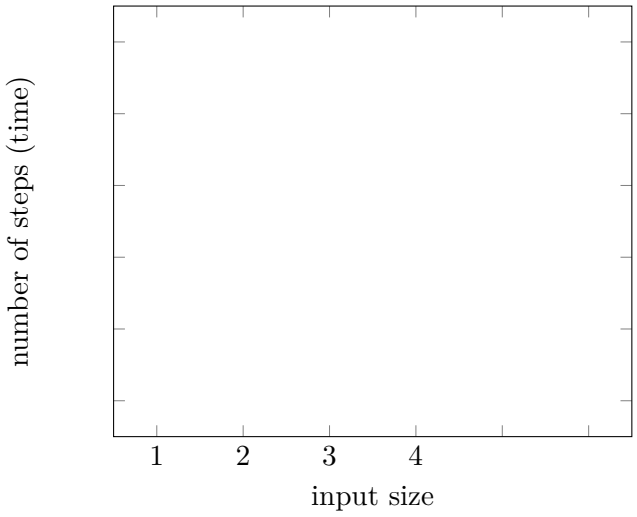
We measure the running time of an algorithm by counting the number of steps it takes to complete.

Remarks:

- The RAM model is not accurate but it can provide rough estimates for the running time of algorithms.
- There are other models of computation (see COSC261).

2.2 Worst-Case, Average-Case, and Best-Case Complexities

An illustration of the worst-case and best-case time functions for a given (unspecified) algorithm:



These cases can be defined more formally. Let

- $\mathcal{X}$ be the set of all possible inputs for a given algorithm;
- $size : \mathcal{X} \to \mathbb{N}$ be a function that measure the size of an input—that is, $size(x)$ is the size of input x; and
- $time : \mathcal{X} \to \mathbb{N}$ measure the time that the given algorithm takes on each input—that is, $time(x)$ is the number of steps (time) the algorithm takes to compute the output for input x.

Then the *worst-case* time function can defined as

$$T_{WC}(n) = \max_{x \in \{x \in \mathcal{X} | size(x) = n\}} time(x) \ .$$

The *best-case* time function can be defined as

$$T_{BC}(n) = \min_{x \in \{x \in \mathcal{X} | size(x) = n\}} time(x) \ .$$

The average case time function can be defined as

$$T_{AC}(n) = \sum_{x \in \{x \in \mathcal{X} | size(x) = n\}} P\{x |\, size(x) = n\}\, time(x) \ .$$

where $P\{x |\, size(x) = n\}$ is the conditional probability density of x. [The average case is usually difficult to obtain and won't be discussed much in this course.]

Question 2.1. Among different cases for complexity, we are often interested in the worst-case. Why?

2.3 Asymptotic Notation

Motivations:

- The exact time complexity of an algorithm (as an expression/function of its input size) can have too much unnecessary details or can be too complicated to analytically obtain.
- We want to be able to measure the complexity of algorithms and compare them disregarding factors such as hardware speed and programming languages.

Asymptotic (Big O) notation is used to specify upper or lower bounds for complexity functions after ignoring multiplicative constants and lower-order terms.

2.3.1 Formal Definitions

$O(g)$ is the set of all functions that grow <u>at most</u> as fast as g (i.e. g is an asymptotic upper bound for them).

$$O(g) = \{f \mid \exists c > 0 : \exists x_0 : \forall x \geq x_0 : 0 \leq f(x) \leq c\, g(x)\}$$

$\Omega(g)$ is the set of all functions that grow <u>at least</u> as fast as g (i.e. g is an asymptotic lower bound for them).

$$\Omega(g) = \{f \mid \exists c > 0 : \exists x_0 : \forall x \geq x_0 : 0 \leq c\, g(x) \leq f(x)\}$$

$\Theta(g)$ is the set of all functions that grow as fast as g (i.e. g is an asymptotic lower and upper bound for them).

$$\Theta(g) = O(g) \cap \Omega(g)$$

2.3.2 Membership Properties

$$f \in O(g) \wedge f \in \Omega(g) \Leftrightarrow f \in \Theta(g)$$

$$f \in O(g) \Leftrightarrow g \in \Omega(f)$$

$$f \in \Theta(g) \Leftrightarrow g \in \Theta(f)$$

2.3.3 Algebraic Properties

$$f_1 \in O(g_1) \wedge f_2 \in O(g_2) \Rightarrow f_1 + f_2 \in O(g_1 + g_2)$$

$$f_1 \in O(g_1) \wedge f_2 \in O(g_2) \Rightarrow f_1 \times f_2 \in O(g_1 \times g_2)$$

$$g_1, g_2 \geq 0 \Rightarrow O(g_1 + g_2) = O(\max(g_1, g_2))$$

These properties (expressed in terms of O), also hold for Ω and consequently for Θ.

2.3.4 Note on usage

The asymptotic notation might appear in two types of mathematical expressions with two slightly different meanings:

1. with an equal sign, for example, $T(n) = O(n^2)$, in which case it must be interpreted as $T(n) \in O(n^2)$; or
2. within an arithmetic expression, for example, $n^2 + \Theta(n)$, in which case it must be interpreted as $n^2 + f(n)$ where f is a function in $\Theta(n)$.

2.4 A visual guide to asymptotic notation

$n!$	$5n!$	$n! + 2^n$	$\ldots$
2^n	5×2^n	2^{n+1}	$\ldots$
$\vdots$	$\vdots$	$\vdots$	$\vdots$
n^3	$n^3 + 2n^2 + 4n + 1$	$5n^3 + 5$	$\ldots$
n^2	$4n^2 - n + 3$	$n(n+1)$	$\ldots$
$n \log n$	$n \log n + n$	$n \log n^2$	$\ldots$
n	$7n + 4$	$n + \log n$	$\ldots$
$\log n$	$\log n + 3$	$\log n^5$	$\ldots$
1	$\log 9$	2^{500}	$\ldots$

- Functions on the same line are *equivalent* (have the same order).
- The table shows only some sample entries from an infinitely large table. Each line has infinitely many members. There are infinitely many classes between each pair of lines.
- Any function on a given line *dominates* all the functions on the lines below it.
- O of some expression is the set of functions on the corresponding line and all the lines below it.
- Ω of some expression is the set of functions on the corresponding line and all the lines above it.
- Θ of some expression is the intersection of O and Ω of that expression, that is, the set of functions on the corresponding line.
- O, Ω or Θ of any function on a line is the same as that of any other function on the same line. For example, $O(n) = O(7n + 4) = O(n + \log n) = \ldots$
- Think of different cases of running time as a vertical band with the worst case on top, the best case at the bottom, and other cases in between. For some algorithms, the band may span over multiple lines, for some others, it may be entirely on one line.

- A tight bound on a function (for example, worst-case complexity) or set of functions (for example, all-input complexity) is more informative.
- There is no correspondence between asymptotic bounds (O, Ω, and Θ) and the three named time complexity functions (T_{WC}, T_{BC}, and T_{AC}); these are separate concepts. For example, the Θ notion can be used to express a tight bound for T_{WC}. In fact, such a bound would be more informative than a bound expressed with only big O (upper and lower bounds vs only an upper bound).
- When working with logarithms, the following properties may be useful:

  $\log ab = \log a + \log b$

  $\log a^b = b \log a$

2.5 Analysis of Efficiency

2.5.1 (Amortised) Time Complexity of Operations in Python

Operation	Comment	Complexity
`=`	simple assignment (with no complex RHS evaluation)	$\Theta(1)$
`+,-,*,/,//,abs`	simple operations on small numbers	$\Theta(1)$
List indexing	to read or write a single element	$\Theta(1)$
List length	Python keeps track of the length (no need to count)	$\Theta(1)$
Getting a list slice	where the length of the slice is k	$\Theta(k)$
List copy	where the length of the list is n	$\Theta(n)$
List append	appending one element at the end (right)	$\Theta(1)$
List pop	popping the right-most element	$\Theta(1)$
List insert or pop	at non-negative index k of a list of length n	$\Theta(n-k)$
Membership test on lists	using the `in` operator	$O(n)$
Membership test on sets	using the `in` operator	$\Theta(1)$
`min, max, sum`	on a list of length n	$\Theta(n)$
Sorting	a list of length n	$O(n \log n)$

2.5.2 Examples of time analysis on Python programs

Analyse the time complexity of the following Python programs.

```
def proc(a, b, c):
  n = len(a)
  for i in range(n):
    for j in range(n):
      c[i][j] = 0
      for k in range(n):
        c[i][j] += a[i][k] * b[k][j]
```

```
def proc(n):
  r = 0
  for i in range(n):
    for j in range(i+1, n-1):
      for k in range(j):
        r += 1
  return r
```

```
def proc(list_a):
  list_b = list_a
  n = len(list_b)
  if n >= 2:
    x = list_a[0] + list_a[1]
  s = sum(list_b)
  list_a.sort()
```

```
  def proc(alist):
    for i in range(len(alist)):
      min_index = i
      for j in range(i+1, len(alist)):
       if alist[j] < alist[min_index]:
         min_index = j
      alist[i], alist[min_index] = alist[min_index], alist[i]
```

```
  def proc(alist):
    for i in range(len(alist)):
      a = alist[i]
      j = i
      while j >= 1 and alist[j-1] > a:
        alist[j] = alist[j-1]
        j -= 1
      alist[j] = a
```

```
def proc(a):
    if a[0] == 0:
        return min(a)
    else:
        return insertion_sort(a)
```

```
def proc(n):
  r = 0
  while n > 1:
    r += 1
    n -= 2
```

```
def proc(n):
  r = 0
  while n > 1:
    r += 1
    n //= 2
```

```
def proc(n):
  while n > 1:
    if n % 2 == 0:
      n //= 2
    else:
      n = 3 * n + 1
```

3 Recursion

Recursion is recurring theme in mathematics and computer science. In this chapter we look at *induction* as a way proving the correctness of some algorithms and *divide-and-conquer* as a design technique.

3.1 Induction

Induction is the most widely-used technique in proving propositions in computer science and software engineering. One reason for this is the vast number of recursive data structures and algorithms.

Suppose $p(n)$ is a logical statement (a claim that could be true or false) involving the natural number $n \in \mathbb{N}$, sometimes called a predicate. The principle of induction states that

$$\forall n_0 : (p(n_0) \wedge (\forall k : p(k) \Rightarrow p(k+1))) \Rightarrow (\forall n \geq n_0 : p(n)) \ .$$

Example 3.1. Suppose $p(n)$ states that $2n - 5 > 0$. It can be seen that $p(3)$ holds (because $2 \times 3 - 5 = 1 > 0$).

Also if $p(k)$ is true, that is $2k-5 > 0$, then $2k-5+2 > 0$ is true, which means $2(k+1)-5 > 0$ is true. Thus, $p(k+1)$ is true. In other words, $p(k) \Rightarrow p(k+1)$.

From these two, based on the principle of induction we conclude that $p(n)$ holds for all $n \geq 3$.

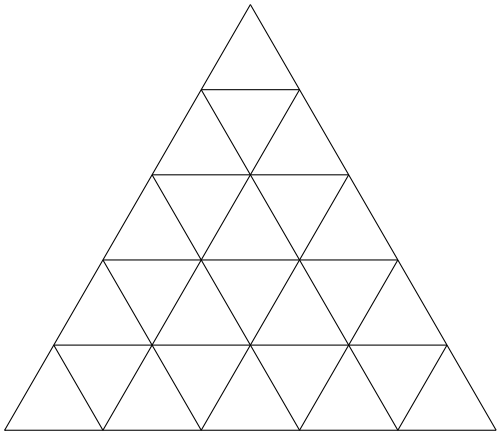
3.1.0.1 Inductive proofs

You can use the principle of induction to prove that a statement $p(n)$ is true for all natural numbers greater than or equal to some n_0. You need to prove two things:

1. **basis (base case)**: prove $p(n_0)$ is true; that is, prove that the statement holds for some natural number n_0 (usually 0 or 1, but it could be any other natural number).
2. **inductive step**: prove $[p(k) \Rightarrow p(k+1)]$ is true; that is, prove that if the statement is true for any k, it will also be true for $k+1$.

A common approach to prove the inductive step is to assume that the statement is true for some k (inductive hypothesis) and then use this assumption to show that the statement is also true for $k+1$.

Example 3.2. Consider a large triangle that contains smaller triangles withe following patter:



Let $p(n)$ be the following statement: "A large triangle of size n contains n^2 small triangles." Prove that $p(n)$ is true for each natural number $n \geq 1$.

Proof. **basis**: $p(1)$ is true.

inductive step: [Show that if $p(k)$ is true, then $p(k+1)$ will be true as well.] Assume a large triangle of size k has k^2 small triangles. Then a large triangle of size $k+1$ has $k^2 + k + k + 1 = k^2 + 2k + 1 = (k+1)^2$ small triangles; that is, $p(k+1)$ is true. □

Example 3.3. Consider the function described below. The function takes a natural number as input.

```
procedure SUM(n)
  s ← 0
  for i from 1 to n
    s ← s + i
  return s
```

Prove that SUM(n) returns $n(n+1)/2$.

Proof. Let $p(n)$ be the following statement: "the value of s after the n-th iteration of the loop is $\frac{n(n+1)}{2}$". Since SUM(n) returns s after n iterations of the loop, to prove that SUM(n) $= n(n+1)/2$, it suffices to prove that $p(n)$ is true for each natural number n.

basis: $p(0)$ is true because before the first iteration of the loop the value of s is 0 (which equals $0(0+1)/2$).

inductive step: if $p(k)$ is true, that is, the value of s after k iterations is

$$\frac{k(k+1)}{2} \ ,$$

then after one more iteration the value of s is

$$\frac{k(k+1)}{2} + (k+1) = \frac{k(k+1) + 2(k+1)}{2} = \frac{(k+1)(k+2)}{2} \ .$$

That is, after $k+1$ iterations the value of s is $(k+1)((k+1)+1)/2$. This means that $p(k+1)$ is true. Therefore, $p(k) \Rightarrow p(k+1)$.

The truth of the basis, $p(0)$, together with the inductive step imply that $p(n)$ is true for each natural number n. □

Example 3.4. Prove the correctness of the following algorithm.

procedure INSERTION-SORT(A, key)
  for i **from** 0 **to** $length(A) - 1$
    $a \leftarrow A[i]$
    $j \leftarrow i$
    while $(j \geq 1) \wedge (key(A[j-1]) > key(a))$
      $A[j] \leftarrow A[j-1]$
      $j \leftarrow j - 1$
    $A[j] \leftarrow a$

Proof. Let $p(n)$ be the following statement: "after one complete iteration of the for-loop, where i is n, the sub-array $A[0..n]$ is in sorted order". We use induction to prove the correctness of $p(n)$, for $n \geq 0$.

basis: $p(0)$ is true because the first iteration of the for-loop (in which $i = 0$), effectively does nothing. Note that the while-loop is not executed. This is because a sub-array of size one is always sorted[1].

inductive step: Let us assume $p(k)$ holds. Then in the next iteration, when $i = k+1$, the elements of the sub-array $A[0..k+1]$, starting from the k-th element, are shifted to the right so that the $(k+1)$-th element is inserted at the correct position[2]. Finally, the last line of the body of the loop inserts the element at the correct position. Therefore $p(k+1)$ holds. □

[1] In fact, the for-loop of a typical Insertion Sort starts from index 1 (the second element of the array). We modified the algorithm here to make the proof straightforward. The modification does not change the correctness or the asymptotic efficiency of the algorithms.

[2] This is a rather informal proof. A more formal proof would use induction for the inner while-loop as well.

3.2 Divide and Conquer

Divide and Conquer (D&C) is an algorithm design technique. In this technique a (large) problem is solved recursively in three steps:

i) **Divide** the problem into a number of subproblems that are smaller instances of the same problem.

ii) **Conquer** the subproblems by solving them recursively.

iii) **Combine** the solutions to the subproblems into the solution for the original problem.

When the subproblems are large enough, they are solved recursively. This is the **recursive case** of the algorithm.

For small enough subproblems, a direct solution (without subdividing) must exist. This forms the **base case** of the algorithm.

Example 3.5. Merge sort is a D&C algorithm defined as follows. It sorts the sub-array $A[l \,..\, r]$.

```
1  procedure MERGE-SORT(A, l, r)
2    if l < r
3      m ← ⌊(l+r)/2⌋
4      MERGE-SORT(A, l, m)
5      MERGE-SORT(A, m + 1, r)
6      MERGE(A, l, m, r)
```

Identify the three steps of D&C for each subproblem. What is the base case?

Divide: Divide the n-element sequence to be sorted into two subsequences of $n/2$ elements each.

Conquer: Sort the two subsequences recursively using merge sort.

Combine: Merge the two sorted subsequences to produce the sorted sequence.

The base case is ...

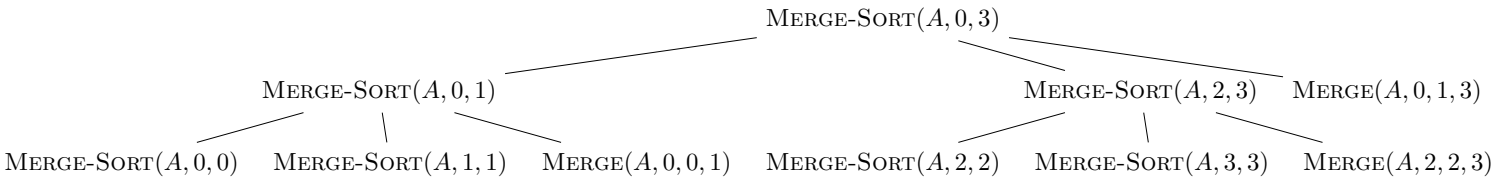
△

3.2.1 Invocation trees

The *invocation tree* of a procedure shows the calls (invocations) that are made by the procedure. It is recursively defined as follows.

1. The root of the tree corresponds to the call to the procedure (with the required arguments).
2. The children of the root (if any) from left to right are invocation trees corresponding to calls made by the root.

Example 3.6. Draw the invocation tree of MERGE-SORT$(A, 0, 3)$.



△

Question 3.1. Consider an invocation tree. In which order are the calls initiated? In which order are they completed?

3.2.2 Analysis of D&C Algorithms

The running time of a recursive algorithm can be expressed by a **recurrence** equation which describes the running time of a problem with a given size in terms of the running times of smaller instances.

In general the running time of a D&C algorithm can be expressed as a function with two cases:

1. Time to solve the smallest problem (base case) = $\Theta(1)$.
2. Time to solve a bigger problem (recursive case) = time to divide + time to conquer + time to combine

Example 3.7. What is the running time recurrence equation of merge sort?

Let $T(n)$ be the time to sort an array with n elements (that is, to solve an instance of size n). Time to divide is $\Theta(1)$. Time to conquer is $2T(n/2)$. Time to combine (merge) is $\Theta(n)$. Therefore,

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

△

The time complexity of a D&C algorithm is not directly visible from its recurrence equation. We need to "solve" the recurrence equation – that is, find a closed-form (non-recursive) expression for the running time.

3.2.3 Solving Recurrence Equations using Recurrence Trees

We use recurrence trees (also called recursion trees) to find a closed-form expression (or asymptotic bound) for the complexity of a D&C algorithm.

A recurrence tree is obtained by expanding the corresponding recurrence equation and has the following properties:

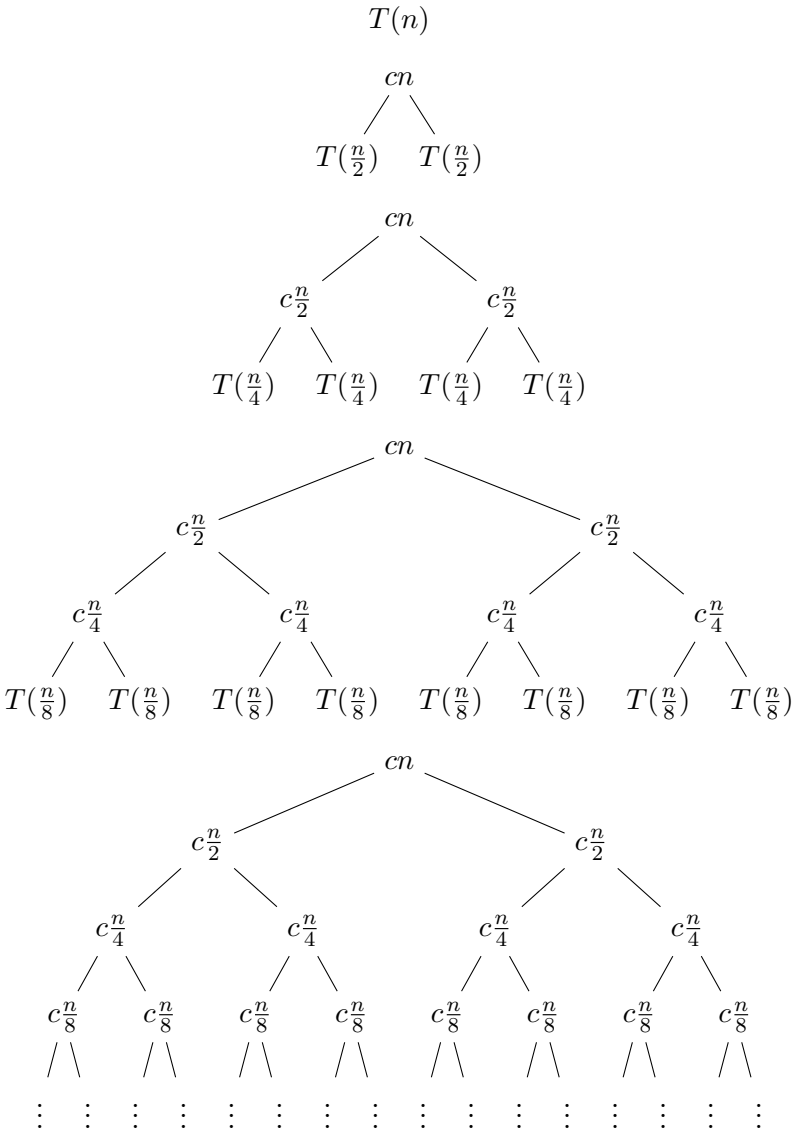
- it is a tree;
- each node is an expression for a cost;
- each subtree corresponds to a subproblem;
- the size of the subproblem of a subtree is smaller than that of its parent (a consequence of D&C design technique);
- the sum of all nodes (including the root) of a subtree is the cost of the corresponding subproblem;
- each subtree in the tree is a recurrence tree for a subproblem.

Therefore the cost of a problem of size n at the root of the tree is the sum of all nodes in the tree. An easy method to find the overall sum is to first sum the costs at each *depth* of the tree and then find the sum of these sums.

The tree is expanded downwards until the base case subproblems are reached. The entire tree need not be drawn. The purpose of the tree is to find a closed-form expression for the cost.

Example 3.8. Draw the recurrence tree of the merge sort algorithm. What are the sums at each depth? What is the sum of the entire tree?

The recurrence tree is shown at various stages of expansion.



- The number of nodes at depth d is 2^d.
- The cost expression of each node at depth d is $c\frac{n}{2^d}$.
- The sum at each depth is $2^d \times c\frac{n}{2^d} = cn$.
- We reach the base case when $\frac{n}{2^d} = 1$; that is, when $d = \log n$.
- The sum of all nodes is $T(n) = cn \log n = \Theta(n \log n)$.

△

Example 3.9. Give a D&C algorithm for solving a Tower of Hanoi problem of size (height) n. Express the corresponding recurrence equation and solve it using recurrence trees.

procedure MOVE-TOWER(*height*, *source*, *destination*, *auxiliary*)
  if *height* ≥ 1
    MOVE-TOWER(*height* -1, *source*, *auxiliary*, *destination*)
    MOVE-DISK(*source*, *destination*)
    MOVE-TOWER(*height* -1, *auxiliary*, *destination*, *source*)

The recurrence equation is

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

The recurrence tree is ...

- The number of nodes at depth d is 2^d.
- The cost expression of each node at depth d is c.
- The sum at each depth is $c2^d$.
- We reach the base case when $n - d = 1$, that is, when $d = n - 1$.
- The sum of the tree is $T(n) = \sum_{d=0}^{n-1} c2^d = c(2^n - 1) = \Theta(2^n)$.

Note that similarly we can define a recurrence equation for the number of disks that need to be moved as follows:

$$M(n) = \begin{cases} 1 & \text{if } n = 1, \\ 2M(n-1) + 1 & \text{if } n > 1. \end{cases}$$

The recurrence tree would be identical except that c is replaced with 1. Therefore the number of disks that need to be moved in order to solve a Tower of Hanoi problem of height n is equal to $2^n - 1$.

△

Example 3.10. Binary search is a D&C algorithm. Given an array A of size n that is sorted based on a function *key* and a value x that we are searching for, it returns an index i such that $key(A[i]) = x$. It returns NULL if there is no such value.

```
1  procedure BINARY-SEARCH(A, key, x, l, r)
2    if l > r
3      return NULL
4    m ← ⌊(l+r)/2⌋
5    if x < key(A[m])
6      return BINARY-SEARCH(A, key, x, l, m − 1)
7    else if x = key(A[m])
8      return m
9    else
10     return BINARY-SEARCH(A, key, x, m + 1, r)
```

Give a tight bound on the worst-case time complexity of this algorithm using recurrence trees.

The recurrence equation is

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

The recurrence tree is ...

- The number of nodes at depth d is 1.
- The cost expression of each node at depth d is c.
- The sum at each depth is c.
- The base case is reached when $\frac{n}{2^d} = 1$, that is, when $d = \log n$.
- The sum of the tree is $T(n) = c \log n = \Theta(\log n)$.

△

3.2.4 Fast Exponentiation

Logarithms can be used to evaluate exponentiations efficiently. The process is efficient because logarithms reduce exponentiation to multiplication. The drawback is limited numerical precision.

In some problems (such as primality testing and cryptography) we need the exact value of an expression like a^n where n can be quite large. A simple algorithm is to multiply a by itself n times. The time complexity of the simple approach is linear (that is, $\Theta(n)$).

Using D&C techniques, this can be computed more efficiently. Observe that if n is even, then $a^n = a^{\frac{n}{2}} a^{\frac{n}{2}}$. If n is odd, then $a^n = a^{\lfloor \frac{n}{2} \rfloor} a^{\lfloor \frac{n}{2} \rfloor} a$. In either case the problem can be reduced to a subproblem half its size. Therefore the following algorithm evaluates the exponentiation efficiently.

```
1  procedure POWER(a, n)
2    if n = 0
3      return 1
4    else
5      p ← POWER(a, ⌊n/2⌋)
6      if n is even
7        return p × p
8      else
9        return a × p × p
```

Question 3.2. What is the time complexity of the fast exponentiation algorithm?

3.2.5 More examples

Example 3.11. Consider the following recursive implementation of the factorial function:

```
procedure FACTORIAL(n)
  if n = 0
    return 1
  else
    return n × FACTORIAL(n − 1)
```

a) Without a formal analysis, do you think there is any difference between the time complexity of this algorithm and an iterative approach?

b) Is there any difference between worst-case and best-case time complexity of the algorithm?

c) How does the recurrence tree look like?

Example 3.12. Let F_n denote the n-th Fibonacci number; that is, $F_0 = 0$ and $F_1 = 1$ and $F_n = F_{n-1} + F_{n-2}$ for $n \geq 2$. You can use induction to show that the following holds:

$$\begin{pmatrix} F_{n+1} & F_n \\ F_n & F_{n-1} \end{pmatrix} = \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}^n.$$

How do you think this equation can be used to devise an efficient D&C algorithm to compute F_n with time complexity $O(\log n)$?

4 Graphs

A graph is an ordered pair (V, E), where V is a set of vertices, E is a set of edges, and each edge relates two vertices together. In the context of the algorithms introduced in this course, we assume that for every pair of vertices, there is at most one edge connecting them[3].

Figure 1 shows the drawing of a graph. In drawings, the shape, location, or size of a vertex and the length of an edge (e.g. the length of the line) do not carry any information. For the sake of readability, graphs must be drawn in such a way that edges do not cross each other but this is not always possible.

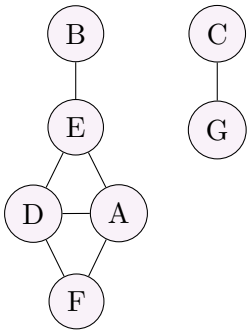


Figure 1: A graph with 7 vertices and 7 edges

We would like to look at graph algorithms in an abstract way. This would enable us to solve a wide range of problems using the same general tool. For example, the graph presented in Figure 1 may be related to a problem in which vertices represent cities and edges represent direct (two-way) flights between pairs of cities. We will see that, for example, a BFS traversal starting from vertex D can be used to find an itinerary with the smallest number of connecting flights from city D to any other city. If the graph in Figure 1 was instead related to a problem in which vertices represented people and edges represented mutual friendship between two people, then the same BFS traversal could have been used to determine the degrees of separation between person D and other people.

4.1 Concepts and terminology

In this section, we look at some graph concepts that are encountered in this course.

[3] We do not study multi-graphs which allow multiple edges between the same pair of vertices.

4.1.1 Size of graphs

We measure the size of a graph in terms of the numbers of vertices and edges. We use n to denote the number of vertices (i.e. $|V| = n$) and m to denote the number of edges (i.e. $|E| = m$). The inequality $m \leq n^2$ always holds because edges connect pairs of vertices, and therefore, there are at most n^2 edges in a graph (which is the case when all pairs of vertices are connected).

We say a graph is *sparse* when $m \in O(n)$. Many real-world graphs (e.g. friendship graphs or road networks) fall in this category. We say a graph is *dense* when $m \in \Theta(n^2)$; that is, a vertex is connected to most other vertices.

4.1.2 Directed and undirected graphs

A graph can be classified as *directed* or *undirected*. In directed graphs, all edges are directed. In undirected graphs, all edges are undirected. In this course we do not consider mixed graphs were some edges are directed and some undirected.

4.1.2.1 Directed graphs

In directed graphs, edges specify one-way relationships. For example, if we want to model a network of roads in which some roads may be one-way, we use a directed graph. If there is a one-way connection from a to b, then the graph will have a directed edge from a to b. If the connection between b and c is two-way, then the graph will have two edges: one from b to c another from c to b.

In directed graphs $E \subseteq V \times V$. This means that edges are *ordered pairs* (two-tuples of the form (u, v) where $u, v \in V$). The edges are ordered because (u, v) (which means $u \to v$) is different from (v, u) (which means $v \to u$).

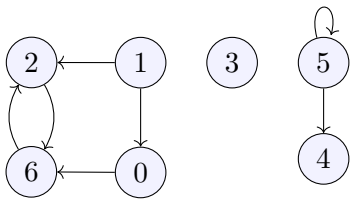


Figure 2: A directed graph $G = (V, E)$, where $V = \{0, 1, \ldots 6\}$ and $E = \{(1, 0), (2, 6), (5, 4), (1, 2), (5, 5), (0, 6), (6, 2)\}$

A directed edge (u, v) can be used to reach vertex v from vertex u but cannot be used to reach vertex u from v. Figure 2 shows an example of a directed graph. Note that directed edges are drawn with arrows.

4.1.2.2 Undirected graphs

In undirected graphs, edges specify two-way (symmetric) relationships. For example, if we want to model a network of roads in which all roads are two-way, we use an undirected graph.

In undirected graphs, $E \subseteq \{\{u, v\} : u, v \in V\}$. This means that edges are *unordered pairs*. The edges are unordered because $\{u, v\}$ (which means

) is the same as $\{v, u\}$. Note that, we are using curly braces to show that $\{u, v\}$ is a set, which implies that the order of elements does not matter. An undirected edge $\{v, u\}$ can be used to reach u from v or to reach v from u; that is, it is equivalent to having two directed edges (v, u) and (u, v) in the directed version of the graph.

Figure 3 shows an example of a directed graph. Note that, undirected edges are drawn as plain line segments (without arrow heads).

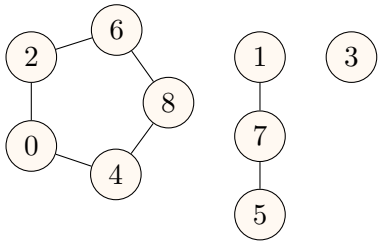


Figure 3: An undirected graph $G = (V, E)$, where $V = \{0, 1, \ldots 8\}$ and $E = \{\{7, 1\}, \{2, 6\}, \{0, 4\}, \{0, 2\}, \{8, 6\}, \{4, 8\}, \{7, 5\}\}$

4.1.3 Weighted graphs

A weighted graph is a triple $(V, E, \mathcal{W})$, where $\mathcal{W} : E \to \mathbb{R}$. The additional component is a function $\mathcal{W}$ that maps each edge to a real value (i.e. assigns a real number to each edge). Weighted graphs can be directed or undirected. Figure 4 shows an example of a weighted undirected graph.

We use weighted graphs to associate a numeric value to each edge and later use this information to find an optimal solution. For instance, in a graph representing a road network, where the vertices are junctions and the edges are road segments, weights can represent the length of road segments. Using a shortest path algorithm, we can find a path from a source vertex to a target vertex with the smallest total weight.

4.1.4 Paths and cycles

Given a graph, we define a *path* to be a sequence of vertices $(v_0, v_1, \ldots, v_k)$ such that

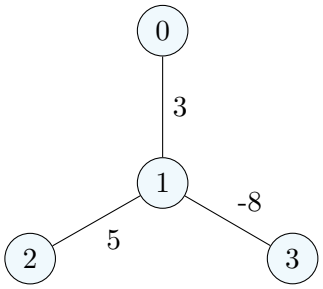


Figure 4: A weighted undirected graph $G = (V, E, \mathcal{W})$, where $V = \{0, 1, 2, 3\}$, $E = \{\{2, 1\}, \{1, 0\}, \{3, 1\}\}$, $\mathcal{W}(\{1, 3\}) = -8$, $\mathcal{W}(\{1, 2\}) = 5$, and $\mathcal{W}(\{0, 1\}) = 3$

i) for each i in $\{0, 1, \ldots, k-1\}$, we have $(v_i, v_{i+1}) \in E$ or $\{v_i, v_{i+1}\} \in E$; and

ii) the path does not use any edge more than once.

We call the first vertex in the path (v_0), *source*, and the last vertex (v_k), *destination*.

The *length* of a path is the number of edges across the path. For instance, the length of the path $(v_0, v_1, \ldots, v_k)$ is k.

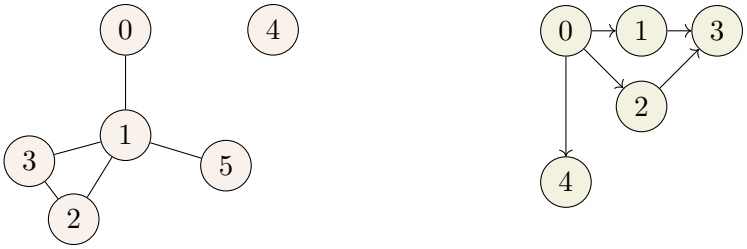


Figure 5: Left: an undirected graph in which $(3, 2, 1, 0)$ is a path of length three; $(2, 3, 1, 2)$ is a path of length three; $(0, 1, 2, 3, 1, 5)$ is a path of length five; $(0, 1, 0)$ is not a path (because the edge $\{0, 1\}$ is used more than once); and the only path that has 4 as its source or destination is (4) which is of length zero. The graph is cyclic. **Right:** a directed graph in which there are two paths from 0 to 3 (both of length two). This graph is a DAG.

A *cycle* is a path of length at least 1 whose source and destination are the same vertex. A graph that contains one or more cycles is called *cyclic*. A graph that does not have any cycle is called *acyclic*. A directed graph that does not have any cycle is called a DAG (directed acyclic graph).

A *loop* or *self-loop* is a (directed or undirected) path of length one that connects a vertex to itself.

See Figure 5 for some examples of paths and cycles.

Remarks:

- Every loop is a cycle.
- There is no undirected graph that has a cycle of length two[4].

4.1.5 Subgraphs and components

4.1.5.1 The empty graph

The graph $(\{\}, \{\})$ is the empty graph. The empty graph does not have any vertices and therefore does not have any edges. The empty graph can be considered to be directed or undirected, weighted or unweighted.

4.1.5.2 Subgraphs

A graph $G' = (V', E')$ is a subgraph of a graph $G = (V, E)$, if $V' \subseteq V$ and $E' \subseteq E$. Figure 6 shows a few examples of subgraphs of a graph.

Remarks:

- The definition implies that a subgraph is a valid graph in its own right.
- For weighted graphs, the weight values remain unchanged in the subgraph.
- Every graph is a subgraph of itself.
- The empty graph is a subgraph of every graph.

4.1.5.3 Components of a graph

Consider an undirected graph $G = (V, E)$. A graph $G' = (V', E')$ is a component of G if:

i) G' is a non-empty subgraph of G;

ii) every pair of vertices in G' are connected by a path in G;

[4] In our definition of graphs, we did not allow parallel edges. If parallel edges were allowed then there could be an undirected cycle of length two when there are two undirected edges between two vertices.

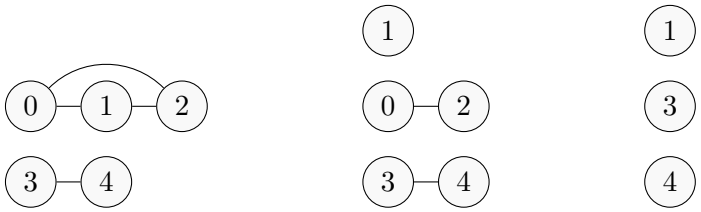


Figure 6: A graph (left) and two (out of many) of its subgraphs (middle and right)

iii) for all $v' \in V'$ and for all $u \in (V \setminus V')$, there is no path between v' and u in G; and

iv) E' contains all edges in E that involve vertices in V'.

In other words, a component of a graph is a maximal connected subgraph. See Figure 7 for an example of a graph with multiple components.

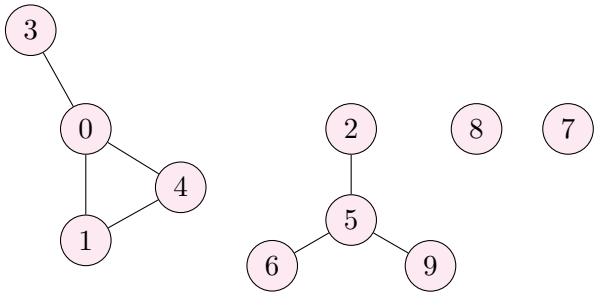


Figure 7: A graph with four components. One component includes vertices 0, 1, 3, 4, and all the edges connecting them. Another component includes vertices 2, 5, 6, 9 and all the edges connecting them. The other two components are vertices 7 and 8.

Since the empty graph does not have any non-empty subgraph, by definition it does not have any components.

4.1.6 Trees and forests

A *tree* is a (directed or undirected) graph in which one vertex is considered to be the *root* and for every vertex in the graph there is one (and only one) path between the vertex and the root[5]. In a directed tree, all the edges are either towards the root or away from it. Figure 8 shows some examples of trees.

[5]In graph theory, a tree does not need to have a distinct root. What we are defining here is known as a *rooted tree* in graph theory.

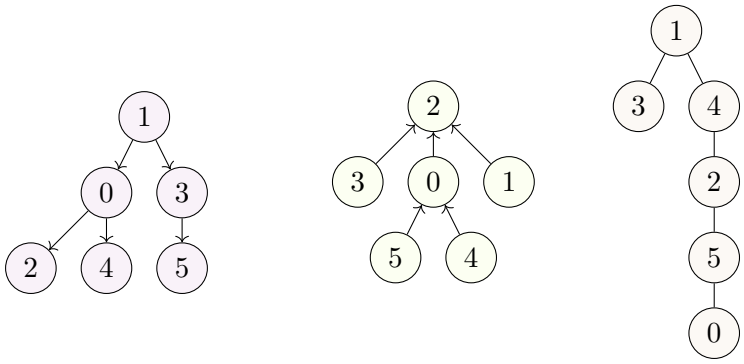


Figure 8: Three examples of trees: two directed trees (left and middle) and one undirected tree (right)

A *forest* is a graph whose every component is a tree. Figure 9 shows an example of a forest.

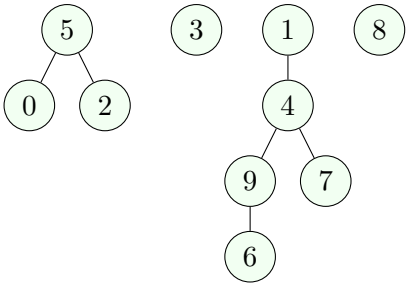


Figure 9: This graph is a forest. The graph has four components all of which are trees. Note that, two of these trees are just single leaves.

Remarks:

- Every graph that is a single vertex is a tree.
- Every tree is a forest.
- Every forest is a graph.

4.2 Textual representation of graph objects

In order for computer programs to be able to take graphs as input, we must define a precise format for graphs. In this course, we use a string-based (text)

format. A textual format (as opposed to binary) allows graph inputs to be easily inspected and edited by humans.

For a graph with n vertices, we assume that the vertices have been assigned natural numbers from 0 to $n - 1$. The assignment of integers to vertices is arbitrary and does not carry any semantics. For example, vertex 9 is not greater (or more important) than vertex 8. We only have to guarantee that there is a one-to-one mapping from $\{0, 1, \ldots, n - 1\}$ to the n vertices in our application domain (e.g. cities in a road network).

4.2.1 Specification

The string will have one or more lines. The first line is the header which describes the graph. The remaining lines (if any) describe the edges of the graph.

Each line has a number of fields separated from each other by one or more blank spaces. There may be spaces before the first field and after the last field. Each line is terminated by a newline character.

4.2.1.1 The header

The header describes the type of the graph and the number of vertices.

The first field is either the character `D` indicating a directed graph, or the character `U` indicating an undirected graph.

The second field of the header is a natural number n indicating the number of vertices in the graph. All the vertices in the graph are numbered from 0 to $n - 1$.

If the graph is weighted, there will be a third field: the character `W`.

4.2.1.2 The edges

If the graph string has more than one line, each remaining line (starting from the second line) describes an edge. The first two fields of an edge line are two natural numbers between 0 and $n - 1$, describing an edge between the two vertices identified with these numbers.

If the graph is directed, the two numbers must be interpreted as an ordered pair describing an edge from the first vertex to the second vertex.

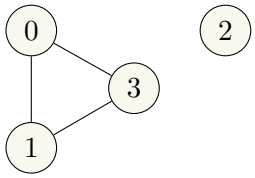
If the graph is undirected, the two numbers must be interpreted as an unordered pair describing an edge between the two vertices. An undirected edge appears only once in the text. For instance, in an unweighted undirected graph, if there is an edge between vertices 0 and 2, the text will contain either the line `0 2` or `2 0` but not both.

If the graph is weighted, there will be a third field which will be an integer indicating the weight of the edge.

Example 4.1. The following text describes an undirected graph (which is also unweighted since there is no `W` in the header) with four vertices and three edges.

```
U 4
0 1
3 1
0 3
```

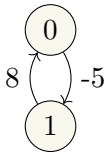
The drawing of the graph is



Example 4.2. The following describes a weighted directed graph with two vertices and two edges.

```
D 2 W
0 1 −5
1 0 8
```

The drawing of the graph is



Example 4.3. The following describes an unweighted directed graph with four vertices and no edge.

```
D 4
```

The drawing of the graph is

4.3 Graph data structures

When a computer program reads the textual description of a graph, it has to parse the input and then construct an appropriate data structure in the RAM so that graph algorithms can work with the graph.

A naive solution would be to construct two sets V and E similar to the formal (mathematical) definition we provided. This is not ideal because some operations become expensive. For example, it would take $\Theta(m)$ time to find edges that are connected to a certain vertex. For a dense graph this amounts to $\Theta(n^2)$.

Two common data structures for graphs are *adjacency lists* and *adjacency matrices*.

4.3.1 Adjacency matrix

An unweighted graph with n vertices can be represented by an $n \times n$ binary matrix A, such that

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

where $a_{i,j}$ is the element in row i and column j.

Remarks:

- For an undirected edge $\{i,j\}$, both $a_{i,j}$ and $a_{j,i}$ are set to 1.
- The adjacency matrix of an undirected graph is symmetric.
- The elements on the main diagonal are 0 unless there is a loop.
- Instead of 0/1 other pairs of symbols (e.g. false/true) can be used.
- For weighted graphs, when there is an edge from i to j, $a_{i,j}$ is the weight of the edge. When there is no edge, $a_{i,j}$ is a special symbol like NULL (in Python `None`).
- For implementing the matrix in Python, one could use NumPy, or a list of size n where each element is a list of size n itself. The elements of the inner lists are 0/1 or weights depending on the type of the graph.

Example 4.4. Consider the following graph.

```
U 4
2 3
1 0
2 1
```

Using a list of lists in Python, the adjacency matrix is:

```
[
  [0, 1, 0, 0],
  [1, 0, 1, 0],
  [0, 1, 0, 1],
  [0, 0, 1, 0],
]
```

Example 4.5. Consider the following graph.

```
D 3 W
1 2 7
1 0 −3
2 1 9
2 2 1
0 2 0
```

Using a list of lists in Python, the adjacency matrix is:

```
[
  [None, None,  0],
  [  -3, None,  7],
  [None,    9,  1],
]
```

4.3.2 Adjacency lists

The idea behind the adjacency list data structure is to have n lists, one for each vertex, where the lists hold edges associated with each vertex. There are different variations of this data structure. The variation we use in this course is an array (list) Adj of length n, where $Adj[i]$ itself is a list containing directed edges from vertex i or undirected edges involving vertex i.

If (and only if) the graph has a directed edge (i, j) or an undirected edge $\{i, j\}$, then the list at $Adj[i]$ contains j. If the edge is weighted, then the list contains (j, w), where w is the weight of the edge connecting i to j.

Remarks The above definition implies the following:

- If the graph contains an undirected edge $\{i, j\}$, then $Adj[i]$ contains j and $Adj[j]$ contains i.
- The inner lists can have different lengths.
- The inner list for a vertex that is not connected to any vertices is empty.

Also note the following:

- In principle, the order of edges in the inner lists does not matter but for the sake of consistency and testing, the elements must appear in the order they appear in the textual representation of the graph.
- When implementing adjacency lists, in order to be able to use the same data structure for both weighted and unweighted graphs, the elements of the inner lists are always of the form (j, NULL), where NULL (or `None` in Python) in this context, indicates there is no weight.
- In some variations of this data structure, the outer collection is a dictionary instead of a list. This would allow identifying vertices with things other than natural numbers.
- In some variations of this data structure, the inner lists are linked lists.

Example 4.6. Consider the following graph.

U 4 W
3 1 −4
1 0 7

The adjacency list of this graph using Python lists is:

```
[
  [(1, 7)],
  [(3, -4), (0, 7)],
  [],
  [(1, -4)]
]
```

Example 4.7. Consider the following graph.

D 5
2 3
0 4
2 4

The adjacency list of this graph using Python lists is:

```
[
  [(4, None)],
  [],
  [(3, None), (4, None)],
  [],
  []
]
```

4.3.3 On the order of edges in a graph

The mathematical representation of a graph (as a set of vertices and a set of edges) does not specify an order for edges (outgoing from a vertex). However, when working with graphs on a machine, there is always an order (even if it is unspecified). The order is determined by the data structures and procedures used to store and retrieve graph objects.

In the context of this course, the order of vertices in the adjacency list (which is based on the order in which edges appear in the textual representation) dictates which vertex to visit first. For example[6], in the graph

```
U 3
0 1
1 2
2 0
```

a DFS starting from 0 produces `[None, 0, 1]` as the parent array. Whereas in the graph

```
U 3
2 0
0 1
1 2
```

which is mathematically the same graph but has a different adjacency list, a DFS starting from 0 yields `[None, 2, 0]`.

4.3.4 Comparison between adjacency matrices and lists

The following table provides a comparison between the space complexity of adjacency matrices and lists and the time complexity of various operations on these two data structures.

Aspect	Adj. matrix	Adj. list
Determine whether there is an edge (i, j) in the graph	$\Theta(1)$	$O(n)$
Adding an edge	$\Theta(1)$	$O(n)$
Deleting an edge	$\Theta(1)$	$O(n)$
Iterating over edges from a given vertex	$\Theta(n)$	$O(n)$
Iterating over all edges	$\Theta(n^2)$	$\Theta(n+m) = \Theta(max(n, m))$
Space	$\Theta(n^2)$	$\Theta(n+m) = \Theta(max(n, m))$

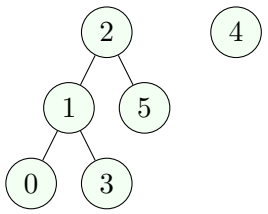
[6]The example makes forward references to the parent array and the DFS algorithm which will be introduced later.

Comments:

- Time for adding an edge in an adj. list is the sum of a time in $O(n)$ (to determine if the edge is already there) and a time in $\Theta(1)$ (to append the edge if necessary).
- Time for deleting an edge in an adj. list is the sum of a time in $O(n)$ (to find the edge) and either a time in $O(n)$ (for deletion in inner lists) or a time in $\Theta(1)$ (for deletion in inner linked lists).
- Since graph traversal involves iterating over all edges and many problems have a sparse graph, the adjacency list is very often the preferred data structure.

4.3.5 Data structure for forests

While we can use adjacency lists or matrices to represent any graph, for forests we can have a more convenient data structure that (as you will see later) makes working with them easier. Recall that, in a forest, every component is a rooted tree. Each vertex in a forest has a single "parent" or has no parent at all. Therefore, the edges of a forest can be described by a function *parent* : $V \to V \cup \text{NULL}$ which maps every vertex to its parent or a special symbol NULL, if the vertex does not have a parent. Since we use natural numbers to refer to vertices, the parent function is of the form *parent* : $\{0, 1, \ldots, n-1\} \to \{0, 1, \ldots, n-1\} \cup \text{NULL}$. Figure 10 provides an example.



$parent(0) = 1$
$parent(1) = 2$
$parent(2) = \text{NULL}$
$parent(3) = 1$
$parent(4) = \text{NULL}$
$parent(5) = 2$

Figure 10: A forest (left) and a function idparent encoding the forest (right)

Some data structures can be viewed as functions. For example, a dictionary can be seen as a function that maps its keys to its values. An array (Python list) is a function that maps natural numbers (from zero up to the length of the list minus one) to values. Since the parent (or predecessor) function of forests is like a lookup table (as opposed to requiring some computation to return a value) and since in some algorithms we want to able to gradually construct a forest (or a tree) as we explore a graph, it is appropriate to use one of these data structures for the parent function. In this course, we use a

parent array of length n, where the value of `parent[v]` is `u`, if $parent(v) = u$. In Python, we use `None` for NULL.

Example 4.8. The code below shows a Python array (list) that represents the parent function in Figure 10. Observe that `parent[i]` in the code has the same value as $parent(i)$ in the figure.

```
parent = [1, 2, None, 1, None, 2]
```

5 Graph traversal

Graph traversal is the process of traversing the edges of a graph in a particular order. The process provides useful information and data structures about the graph that can be used in various applications (for example, finding the number of components, shortest paths, and topological orderings).

5.1 Traversal concepts

5.1.1 States of vertices during traversal

In the traversal process, some (or all) of the vertices of the graph go through the following stages in order:

1. UNDISCOVERED: the vertex has not been encountered yet;
2. DISCOVERED : the vertex has been encountered (discovered), but the algorithm has not finished processing it yet;
3. PROCESSED: the algorithm has finished processing the vertex.

Initially all the vertices in the graph are undiscovered. The traversal starts from a given vertex which will be the first one to be discovered. A traversal algorithm follows the edges going out of discovered vertices (in a certain order) to discover more vertices. Once the traversal algorithm discovers all the vertices connected to a vertex, it marks the vertex as processed.

A graph traversal algorithm needs one or more data structures to keep track of the states of vertices during traversal. In this course, where vertices are identified by numbers $\{0, 1, \ldots, n-1\}$, one could use an array (list) *state* of size n, where each element is either UNDISCOVERED, DISCOVERED, or PROCESSED (or alternatively characters `U`, `D`, or `P`). Another alternative is to have two Boolean arrays *discovered* and *processed* each of size n. This would provide four states for each vertex, where one of them would be invalid (because $discovered[i]$ cannot be false if $processed[i]$ is true).

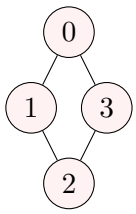
5.1.2 Predecessor tree

One of the data structures that is generated during a graph traversal is the *predecessor tree*. The root of the tree is the vertex at which the traversal starts. While processing vertex u, if we encounter an edge (u, v) (or $\{u, v\}$) such that v is undiscovered, the edge is added to the tree (which makes v a child of u). Those edges of the graph that appear in the tree are called *tree edges* and those that do not appear in the tree are called *non-tree edges*. The predecessor tree can be represented using the *parent* function. If $parent(v) = u$, then the edge (u, v) is a tree edge and u is the parent of v in the predecessor tree.

Example 5.1. Consider the following graph.

```
U 4
3 0
1 2
2 3
0 1
```

Suppose we perform a breadth-first traversal starting at vertex 0. First vertex 0 is discovered. Then vertices 1 and 3 are discovered, which make $\{0, 1\}$ and $\{0, 3\}$ tree edges. At this point, vertex 0 is processed. Next, from vertex 3, vertex 2 is discovered. This makes $\{3, 0\}$ a tree edge. In the next steps, 3, 1, and 2 are marked as processed. Figure 11 visualises the result.




```
[None, 0, 3, 0]
```

Figure 11: Running BFS on this undirected graph starting at vertex 0 produces the parent array on the right.

5.2 Breadth-first search

In BFS, vertices are discovered in order of increasing distance from the starting vertex. We break the BFS algorithm into two procedures:

- BFS-Tree: initialises the required data structures and calls the main loop of the algorithm;
- BFS-Loop: is the main loop of the algorithm and can be reused in BFS applications.

The traversal starts from a starting vertex s.

```
1  procedure BFS-Tree(Adj, s)
2    n ← number of vertices (the length of Adj)
3    state ← an array of length n filled with Undiscovered
4    parent ← an array of length n filled with null
5    Q ← an empty queue
6    state[s] ← Discovered
7    Enqueue(Q, s)
8    return BFS-Loop(Adj, Q, state, parent)
```

```
1  procedure BFS-Loop(Adj, Q, state, parent)
2    while Q is not empty
3      u ← Dequeue(Q)
4      for v in Adj[u]
5        if state[v] = Undiscovered
6          state[v] ← Discovered
7          parent[v] ← u
8          Enqueue(Q, v)
9      state[u] ← Processed
10   return parent
```

5.2.1 Analysis

The size of an input graph $G = (V, E)$ is measured by two metrics: $|V| = n$ and $|E| = m$. Recall that $m \leq n^2$.

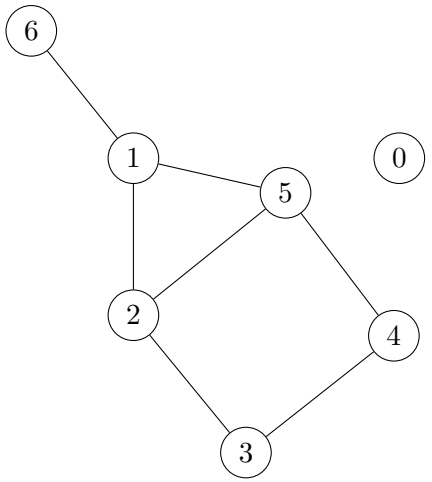
The time complexity of BFS-Tree up until BFS-Loop is called, is $\Theta(n)$.

The time complexity of BFS-Loop is proportional to the number of times the body of the inner loop is executed which is at most as many as the number of edges in the graph, that is, $O(m)$.

Therefore the complexity of BFS-Tree is in $O(n + m) = O(\max(n, m))$.

Example Trace of BFS

```
graph_str = """\
U 7
1 2
1 5
1 6
2 3
2 5
3 4
4 5
"""
```




```
adj_list = adjacency_list(graph_str)

0 [[],
1  [(2, None), (5, None), (6, None)],
2  [(1, None), (3, None), (5, None)],
3  [(2, None), (4, None)],
4  [(3, None), (5, None)],
5  [(1, None), (2, None), (4, None)],
6  [(1, None)]]

bfs_tree(adj_list, 1)
```

	State							Parent						
	0	1	2	3	4	5	6	0	1	2	3	4	5	6
deque([1])	'U'	'D'	'U'	'U'	'U'	'U'	'U'	None	None	None	None	None	None	None
deque([2, 5, 6])	'U'	'P'	'D'	'U'	'U'	'D'	'D'	None	None	1	None	None	1	1
deque([5, 6, 3])	'U'	'P'	'P'	'D'	'U'	'D'	'D'	None	None	1	2	None	1	1
deque([6, 3, 4])	'U'	'P'	'P'	'D'	'D'	'P'	'D'	None	None	1	2	5	1	1
deque([3, 4])	'U'	'P'	'P'	'D'	'D'	'P'	'P'	None	None	1	2	5	1	1
deque([4])	'U'	'P'	'P'	'P'	'D'	'P'	'P'	None	None	1	2	5	1	1
deque([])	'U'	'P'	'P'	'P'	'P'	'P'	'P'	None	None	1	2	5	1	1

Note: The first line shows the value of the variables before entering the while-loop (after initialisation). The remaining lines show the values after completion of each iteration of the while-loop. In the example above, the while-loop has iterated 6 times.

5.3 Depth-first search (DFS)

Similar to BFS, we break the DFS algorithm into two procedures: DFS-Tree and DFS-Loop. While in BFS we use a queue to achieve FIFO (first in, first out) in visiting vertices, in DFS we use a (call) stack to achieve LIFO (last in, first out) and thus proceed to the deepest discovered vertex first.

The traversal starts from a starting vertex s.

```
1  procedure DFS-Tree(Adj, s)
2    n ← number of vertices (the length of Adj)
3    state ← an array of length n filled with Undiscovered
4    parent ← an array of length n filled with null
5    state[s] ← Discovered
6    DFS-Loop(Adj, s, state, parent)
7    return parent
```

```
1  procedure DFS-Loop(Adj, u, state, parent)
2    for v in Adj[u]
3      if state[v] = Undiscovered
4        state[v] ← Discovered
5        parent[v] ← u
6        DFS-Loop(Adj, v, state, parent)
7    state[u] ← Processed
```

Each call to DFS-Loop starts a DFS traversal. Changes to *state* and *parent* are visible across calls (in programming terms, they are references to mutable objects).

The for-loop in DFS-Loop considers edges of the form (u, v). We are interested in a vertex v that has not been discovered before. A DFS is then started from v.

5.3.1 Analysis

The time complexity of DFS-Tree up until DFS-Loop is called, is in $\Theta(n)$.

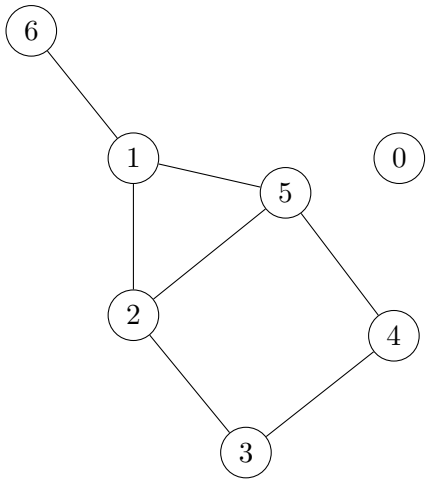
DFS-Loop is called at most n times because it is called when a vertex is discovered and each vertex can be discovered only once.

If DFS-Loop is called on a vertex u, then the for-loop iterates as many times as the number of outgoing edges from u. Therefore the total number of times the body of the for-loop is executed (for all vertices) is in $O(m)$.

Therefore the time complexity of DFS-Tree is in $O(n+m) = O(\max(n, m))$.

Example Trace of DFS

```
graph_str = """\
U 7
1 2
1 5
1 6
2 3
2 5
3 4
4 5
"""
```




```
adj_list = adjacency_list(graph_str)

0 [[],
1  [(2, None), (5, None), (6, None)],
2  [(1, None), (3, None), (5, None)],
3  [(2, None), (4, None)],
4  [(3, None), (5, None)],
5  [(1, None), (2, None), (4, None)],
6  [(1, None)]]

dfs_tree(adj_list, 1)
```

Call-Stack	State							Parent						
	0	1	2	3	4	5	6	0	1	2	3	4	5	6
[1]	'U'	'D'	'U'	'U'	'U'	'U'	'U'	None	None	None	None	None	None	None
[1, 2]	'U'	'D'	'D'	'U'	'U'	'U'	'U'	None	None	1	None	None	None	None
[1, 2, 3]	'U'	'D'	'D'	'D'	'U'	'U'	'U'	None	None	1	2	None	None	None
[1, 2, 3, 4]	'U'	'D'	'D'	'D'	'D'	'U'	'U'	None	None	1	2	3	None	None
[1, 2, 3, 4, 5]	'U'	'D'	'D'	'D'	'D'	'D'	'U'	None	None	1	2	3	4	None
[1, 2, 3, 4, 5]	'U'	'D'	'D'	'D'	'D'	'P'	'U'							
[1, 2, 3, 4]	'U'	'D'	'D'	'D'	'P'	'P'	'U'							
[1, 2, 3]	'U'	'D'	'D'	'P'	'P'	'P'	'U'							
[1, 2]	'U'	'D'	'P'	'P'	'P'	'P'	'U'							
[1, 6]	'U'	'D'	'P'	'P'	'P'	'P'	'D'	None	None	1	2	3	4	1
[1, 6]	'U'	'D'	'P'	'P'	'P'	'P'	'P'							
[1]	'U'	'P'	'P'	'P'	'P'	'P'	'P'							

Note: In this trace, the state and parent arrays are printed before entering the for-loop.
Additionally, the state array is printed right before (a call to) the DFS function returns.

5.4 Properties and applications of BFS and DFS

Prior to looking at some of the applications of BFS and DFS note the following:

- BFS and DFS can operate on both directed and undirected graphs.
- BFS and DFS can be used on weighted graphs. They simply ignore the weight information.
- DFS can be implemented by replacing the queue in BFS with a stack and replacing the Enqueue and Dequeue operations with Push and Pop respectively.
- In an undirected graph, after a call to BFS-Tree or DFS-Tree is finished, all the vertices that are in the same component as the starting vertex are marked as processed. In a directed graph, after the call is finished, all the vertices reachable from the starting vertex are marked as processed.
- A call to BFS-Tree or DFS-Tree creates a single tree represented by the parent array. Even though the parent array can represent any forest, a single call to BFS-Tree or DFS-Tree only creates a single tree.
- Traversal can be performed in an order that is neither depth-first nor breadth-first. The reason BFS and DFS are predominant traversal algorithms is that they have properties that are useful for various applications.
- In BFS, vertices are discovered and processed in order of increasing distance from the starting vertex.
- In DFS, a discovered vertex is marked as processed only after all of its undiscovered adjacent vertices are discovered and processed. In other words, a vertex remains in the discovered state until all the vertices reachable from that vertex are discovered and processed.

5.4.1 Shortest paths

In BFS, vertices are discovered in order of increasing distance from the starting vertex. Therefore, if the root of the predecessor tree is vertex s, and a node in the tree is vertex t, then the unique path from s to t in the tree, is a path with the smallest number of edges from s to t in the graph.

The following algorithm takes a predecessor tree and constructs a path from s to t. Note that, this is applicable when the tree is generated by starting a BFS at s.

procedure TREE-PATH($parent, s, t$)
  if $s = t$
    return the single-vertex path s
  else
    return the path TREE-PATH($parent, s, parent[t]$) followed by t

Example 5.2. Consider the following parent array which represents a predecessor tree constructed by doing a BFS starting at vertex 1 of a graph.

```
[None, None, 1, 2, 5, 1, 1 ]
```

The shortest path from 1 to 4 is:

TREE-PATH($parent, 1, 4$) =
TREE-PATH($parent, 1, 5$) followed by 4 =
TREE-PATH($parent, 1, 1$) followed by 5 followed by 4 =
1 followed by 5 followed by 4.

5.4.2 Connected components

Recall that components (aka connected components) of an undirected graph are maximal subgraphs in which all vertices are reachable from each other.

When BFS or DFS is executed from a starting vertex s, all vertices reachable from s will be discovered and processed. Thus the set of vertices that have been processed forms a component. This process can be repeated until all the vertices in the graph are processed (and consequently all the components are identified). This leads to the following algorithm which uses BFS to find the vertices of the components of an undirected graph.

```
procedure CONNECTED-COMPONENTS(Adj)
  n ← number of vertices (the length of Adj)
  state ← an array of length n filled with UNDISCOVERED
  Q ← an empty queue
  components ← {}
  for i from 0 to n − 1
    if state[i] = UNDISCOVERED
      previous-state ← state
      state[i] ← DISCOVERED
      ENQUEUE(Q, i)
      BFS-LOOP(Adj, Q, state)
      new-component ← all vertices whose state has changed
      components ← components ∪ {new-component}
  return components
```

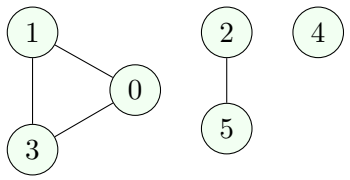
Comments:

- The order of the 'for' loop does not change the functionality of the algorithm but may lead to a different order of discovering components.
- The time complexity of this algorithm is $\Theta(n^2)$ because of the 'for' loop and the line that finds vertices whose state has changed.
- This algorithm can be improved to have the same time complexity as the graph traversal. One way of achieving this is to have an array that maintains a component number for each vertex. The component number can start from 1, for example, and it is incremented inside the loop every time an undiscovered vertex is encountered. The component number is then passed onto BFS which assign it to vertices as they are processed.

Example 5.3. Consider the following undirected graph.

```
U 6
3 0
2 5
1 0
1 3
```

The drawing of the graph is as follows.



The following is the trace of Connected-Components on this graph. The last two columns show the values of *state* and *components* after the iteration.

iteration	call	*state*	*components*
(initialisation)		`[U, U, U, U, U, U]`	{}
0	BFS-Loop(0)	`[P, P, U, P, U, U]`	{{0,1,3}}
1		`[P, P, U, P, U, U]`	{{0,1,3}}
2	BFS-Loop(2)	`[P, P, P, P, U, P]`	{{0,1,3},{2,5}}
3		`[P, P, P, P, U, P]`	{{0,1,3},{2,5}}
4	BFS-Loop(4)	`[P, P, P, P, P, P]`	{{0,1,3},{2,5},{4}}
5		`[P, P, P, P, P, P]`	{{0,1,3},{2,5},{4}}

5.4.3 Testing strong connectivity

A directed graph is *strongly connected* if and only if there is a path between every ordered pair of vertices. In some applications, having a strongly connected graph is important. For example, if a graph represents a road network in which some roads are one-way, then the network should be strongly connected, otherwise it would be possible to go to a point and not be able to come back. Figure 12 shows two examples of strongly connected graphs.

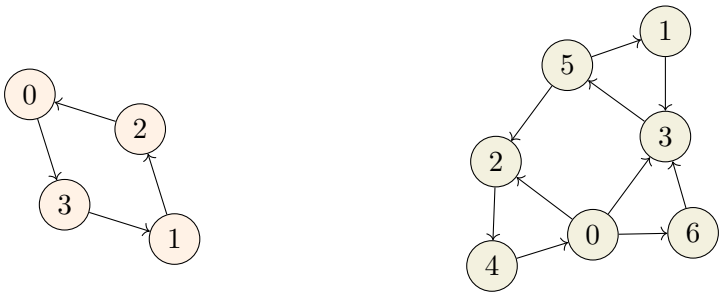


Figure 12: Two examples of strongly connected graphs. In each graph, for every pair of vertices x and y, there is a path from x to y and there is a path from y to x.

A simple algorithm to determine whether a directed graph is strongly connected is to run a traversal algorithm n times, each time starting at a different vertex. If after a traversal (BFS or DFS), all the vertices in the graph are processed, then all vertices are reachable from the starting vertex of the traversal. If this happens after all the n traversals, then all vertices are reachable from each other, and therefore, the graph is strongly connected. The run-time complexity of this algorithm is $O(n(n+m))$.

A more efficient algorithm can be devised. For this purpose, we introduce two new concepts: the *transpose* of a graph, and *hub*s.

Consider a directed graph G. The graph G^T is called the *transpose* or *reverse* of G if it is G with all edges reversed. More formally, given $G = (V, E)$ and $G^T = (V, E^T)$, if $(u, v) \in E$, then $(v, u) \in E^T$, and vice versa. The definition implies that if there is a path from x to y in G, then there is a path from y to x in G^T, and vice versa. It also follows that the transpose of G^T is G. Figure 13 shows an example of transpose.

If G is represented by an adjacency matrix, then G^T can be obtained by transposing G; that is, by reflecting the elements of the matrix along the main diagonal. This operation takes $\Theta(n^2)$ time. If G is represented by an adjacency list, transposing the graph involves looking at all the vertices and reversing edges going out of each vertex. This operation takes $\Theta(n+m)$ time.

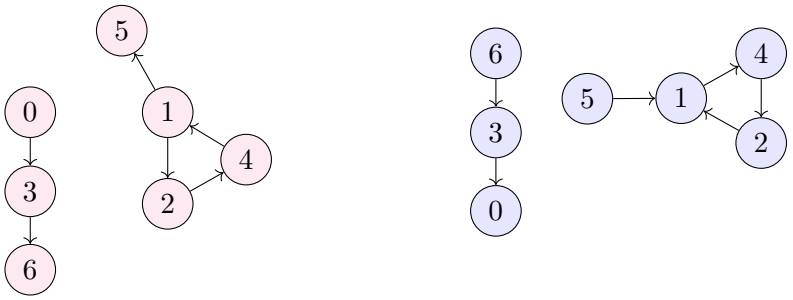


Figure 13: Two graphs that are the transpose of each other. Observe that for every path in one graph, there is a reverse path in the other graph. For example, one graph contains the path $(2, 4, 1, 5)$ and the other contains the path $(5, 1, 4, 2)$.

We define a vertex h to be a *hub* if: (a) every vertex is reachable from h; and (b) h is reachable from every vertex. If a graph has a hub h, then it is strongly connected because, for every pair of vertices x and y, x is connected to h and h is connected to y; thus x is connected to y. For the same reason, if a graph has a hub, then every vertex is a hub. Therefore, to test strong connectivity, it suffices to test whether an arbitrary vertex in the graph is a hub. This leads to the following algorithm which takes a graph G and an arbitrary vertex h, and determines whether the graph is strongly connected:

1. Run a graph traversal (BFS or DFS) on G starting at h;
2. If any vertex remains undiscovered, return False;
3. Construct a graph G^T which is the transpose of G;
4. Run a graph traversal (BFS or DFS) on G^T starting at h;
5. If any vertex remains undiscovered, return False, otherwise return True.

Comments:

- If after any of the two traversals, one or more vertices remain undiscovered, then we have found a pair of vertices that do not satisfy the condition of strong connectivity.
- The run time complexity of the algorithm is the same as graph traversal; that is, $O(n + m)$.

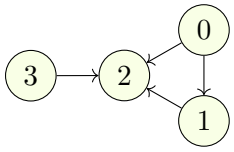
5.4.4 Topological sorting

A *topological ordering* of a directed graph is an ordering of its vertices, such that, if there is an edge (u, v) in the graph, then vertex u is placed in some position before vertex v. Any Directed Acyclic Graph (DAG) has at least one topological ordering.

Example 5.4. Consider the following graph.

```
D 4
0 2
0 1
1 2
3 2
```

The following is the drawing of the graph.



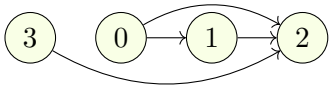
Any of the following is a topological ordering:

$3, 0, 1, 2$

$0, 3, 1, 2$

$0, 1, 3, 2$

The visual interpretation of a topological ordering is an arrangement of the vertices of the graph on a horizontal line such that all the edges point in the same direction (left or right). For example, the first topological ordering listed above can be visualized as follows.



The following includes some examples of topological sorting on DAGs.

- In software engineering, build automation systems (e.g. Make and Apache Maven) apply topological sort to the DAG of dependencies between software components to find a valid order of building software.

- Package management systems (e.g. dpkg, RPM, and Homebrew) apply topological sort to the DAG of package dependencies to find a valid order of installation.
- A prerequisite graph of courses at a university is a DAG. A topological ordering allows doing courses in some valid order.
- Copying relational databases must be done according to a topological ordering of tables (where dependencies are based on foreign keys) so that the referential integrity of the data is maintained during the operation.

Example 5.5. Consider the following graph.

```
D 4
```

The graph does not have any edges. Any ordering of the vertices is a valid topological ordering. There are 4! such orderings.

Example 5.6. Consider the following graph.

```
D 4
0 1
1 2
2 3
3 0
```

The graph has a cycle (i.e. it is not a DAG), and therefore, does not have any topological ordering.

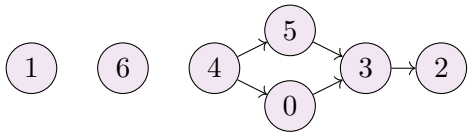
Given a DAG, the following algorithm generates a topological ordering:

1. Initialise the state and parent arrays as usual;
2. Initialise an empty stack;
3. Iterate over all vertices of the graph:
    - If the vertex is undiscovered, start a modified version of DFS-Loop on the vertex. In the modified version, as soon as a vertex is processed, it is pushed onto the stack;
4. Return the content of the stack from top to bottom as a topological ordering.

Example 5.7. Consider the following graph.

```
D 7
0 3
4 0
5 3
3 2
4 5
```

The drawing of the graph is:



The following table shows the trace of the topological sorting algorithm on this graph. The last two columns show the values of *state* and *stack* after each iteration. We use a Python list for the stack; items are pushed and popped on the right. Note that, the trace does not show the call stack.

iteration	call	*state*	*stack*
	(initialisation)	`[U, U, U, U, U, U, U]`	`[]`
0	DFS-Loop(0)	`[P, U, P, P, U, U, U]`	`[2, 3, 0]`
1	DFS-Loop(1)	`[P, P, P, P, U, U, U]`	`[2, 3, 0, 1]`
2		`[P, P, P, P, U, U, U]`	`[2, 3, 0, 1]`
3		`[P, P, P, P, U, U, U]`	`[2, 3, 0, 1]`
4	DFS-Loop(4)	`[P, P, P, P, P, P, U]`	`[2, 3, 0, 1, 5, 4]`
5		`[P, P, P, P, P, P, U]`	`[2, 3, 0, 1, 5, 4]`
6	DFS-Loop(6)	`[P, P, P, P, P, P, P]`	`[2, 3, 0, 1, 5, 4, 6]`

Some lines in the table involve recursive calls that are not shown. At the beginning of iteration 0, vertex 0 is undiscovered; therefore, the modified version of DFS-Loop is called on 0. At this point, vertex 0 is discovered. In the 'for' loop inside DFS-Loop, the undiscovered vertex 3 is encountered, which leads to calling DFS-Loop (recursively) on vertex 3, which then leads to calling DFS-Loop on vertex 2. Vertex 2 does not have any outgoing edges; therefore, the vertex is marked as processed, 2 is pushed onto the stack, and then the recursive call is finished. Next, the recursive call on vertex 3 resumes but there are no more edges; so vertex 3 is also marked as processed and pushed onto the stack. Then, the DFS-Loop call on 0 resumes, which leads to marking 0 as processed and pushing it onto the stack. At the end of iteration 0, the stack contains `[2, 3, 0]` (where the top of the stack is on the right). Some iterations in the table (e.g. iteration 2) do not involve any calls to DFS because the vertex is already processed.

Why does it work? The topological sorting algorithm gradually pushes vertices onto a stack, and at the end, returns the content of the stack from top to bottom. The definition of topological ordering requires that if the graph has an edge (u, v), then u must appear in some position before v in the ordering. Therefore, we must show that, if there is an edge (u, v) in the graph, the algorithm pushes v onto the stack before u.

Inside DFS, when the edge (u, v) is considered, u is discovered but v could be either undiscovered, discovered, or processed:

- If v is undiscovered, then it becomes discovered and a new DFS call starts at vertex v. This means that the call DFS-Loop(v) finishes before DFS-Loop(u); that is, v is processed before u, and therefore, v is pushed onto the stack before u.
- If v is discovered, that means that v is somewhere on the call stack; that is, we have found a path from v to itself. This is not possible because DAGs cannot have cycles.
- If v is processed, then it has already been pushed onto the stack, while u has not yet been pushed.

5.4.5 Cycle detection

Sometimes we want to know if a graph has a cycle. This is, for example, useful to determine whether a directed graph is a DAG, and thus, has a topological ordering.

During DFS traversal, when examining the outgoing edges of a vertex u, if the edge (u, v) goes to a vertex v that is already discovered (that is, it is on the call stack), then the graph has a cycle.

For undirected graphs, the process is similar, but there is one exception: when examining the outgoing edges of a vertex u, we ignore the edge (u, v) where v is the parent of u (v is guaranteed to be discovered but that doesn't count as a cycle because a cycle has to be a path, and a path is not allowed to use an edge more than once).

For undirected graphs with multiple components, or for directed graphs, multiple DFS calls may need to be made in order to find a cycle. Therefore, similar to finding the components of a graph or topological ordering, a for loop must check the state of each vertex in the graph, and run a DFS from that vertex if it is undiscovered.

6 Weighted graph algorithms

6.1 Minimum spanning trees

A *spanning tree* of an undirected graph is a subgraph that is a tree and includes all the vertices of the graph. A *minimum spanning tree* (MST) of a weighted undirected graph is a spanning tree that has the lowest total weight among all other spanning trees. Figure 14 shows a weighted undirected graph and one of its MSTs.

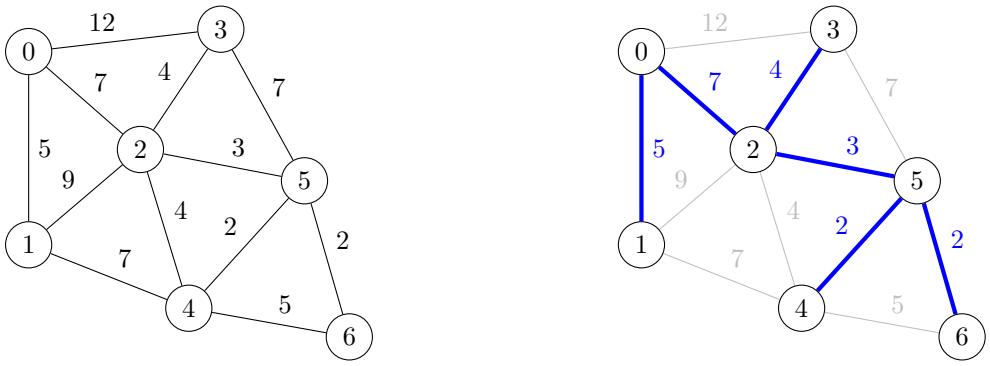


Figure 14: A weighted undirected graph (left) and one of its minimum spanning trees (right). MSTs are not necessarily unique. For this graph, another MST can be obtained by removing the edge $\{0, 2\}$ from the current MST and adding the edge $\{1, 4\}$. All MSTs of a graph have the same total weight.

6.1.1 Prim's algorithm

Given a weighted undirected graph with one component, Prim's algorithm finds a minimum spanning tree. The algorithm is called with an arbitrary vertex as the starting point which forms a one-vertex tree. The tree gradually grows in each iteration by adding the smallest edge between a vertex that is part of the tree and a vertex that is not.

Running Prim's algorithm on vertex 0 of the graph in Figure 14 starts with a tree which initially has only one vertex, 0. The algorithm then looks at all the edges that connect a vertex in the tree to a vertex out of the tree. These edges are $\{\{0, 1\}, \{0, 2\}, \{0, 3\}\}$. Among these, $\{0, 1\}$ with the weight of 5 is the smallest, and therefore, is added to the tree. Next, the algorithm looks at edges $\{\{0, 2\}, \{0, 3\}, \{1, 2\}, \{1, 4\}\}$. At this stage, the algorithm can pick either $\{0, 2\}$ or $\{1, 4\}$ which are equally good. The algorithm continues until all vertices of the graph become part of the tree.

```
1  procedure PRIM(Adj, s)
2    n ← number of vertices (the length of Adj)
3    in-tree ← an array of length n filled with FALSE
4    distance ← an array of length n filled with ∞
5    parent ← an array of length n filled with NULL
6    distance[s] ← 0
7    while ¬all(in-tree)
8      u ← NEXT-VERTEX(in-tree, distance)
9      in-tree[u] ← TRUE
10     for v, weight in Adj[u]
11       if ¬ in-tree[v] ∧ weight < distance[v]
12         distance[v] ← weight
13         parent[v] ← u
14   return parent, distance
```

In the algorithm:

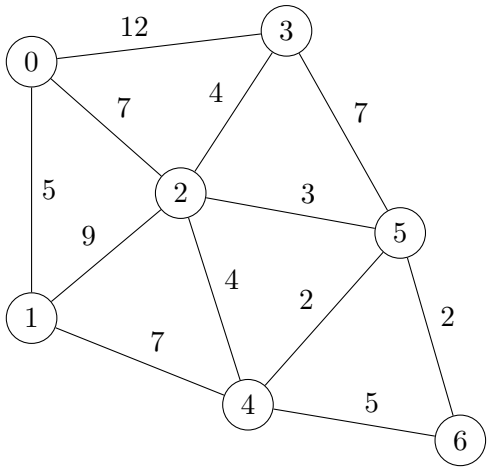
- the *in-tree* array shows which vertices are currently part of the growing tree;
- the *distance* array contains the current distance of each vertex to the growing tree;
- the *parent* array represents the structure of the tree;
- The function NEXT-VERTEX(*in-tree*, *distance*) returns a closest vertex that is not yet part of the tree; and
- the variable *weight* is the weight of the edge (u, v) that is being considered.

Analysis. In a graph with n vertices and m edges, the while-loop iterates n times until all vertices are part of the tree. The call NEXT-VERTEX(*in-tree*, *distance*) takes time proportional to n. Hence, the time inside the while-loop, but outside the for-loop, is $O(n^2)$. The time inside the for-loop over all iterations of the while-loop is $O(m)$ as every edge is considered at most once (as in BFS). Thus, the total time complexity is in $O(\max(m, n^2)) = O(n^2)$ because $m \leq n^2$.

If the distance array is replaced with a more suitable data structure, such as a heap-based priority queue, the time to find the next vertex reduces to $\log n$. Then, the time complexity of the entire algorithm will be in $O(m + n \log n)$. Such a priority queue should support an 'update' operation (because of line 12); otherwise, further modifications must be done.

Example trace of Prim's algorithm

```
graph_string = """\
U 7 W
0 1 5
0 2 7
0 3 12
1 2 9
2 3 4
1 4 7
2 4 4
2 5 3
3 5 7
4 5 2
4 6 5
5 6 2
"""
```




```
adj_list = adjacency_list(graph_string)

0 [[(1, 5), (2, 7), (3, 12)],
1  [(0, 5), (2, 9), (4, 7)],
2  [(0, 7), (1, 9), (3, 4), (4, 4), (5, 3)],
3  [(0, 12), (2, 4), (5, 7)],
4  [(1, 7), (2, 4), (5, 2), (6, 5)],
5  [(2, 3), (3, 7), (4, 2), (6, 2)],
6  [(4, 5), (5, 2)]]

prim(adj_list, 0)
```

u	Distance of vertex i to tree							Parent							In-tree						
	0	1	2	3	4	5	6	0	1	2	3	4	5	6	0	1	2	3	4	5	6
	[0,	*,	*,	*,	*,	*,	*]	[–,	–,	–,	–,	–,	–,	–]	[F,	F,	F,	F,	F,	F,	F]
0	[0,	5,	7,	12,	*,	*,	*]	[–,	0,	0,	0,	–,	–,	–]	[T,	F,	F,	F,	F,	F,	F]
1	[0,	5,	7,	12,	7,	*,	*]	[–,	0,	0,	0,	1,	–,	–]	[T,	T,	F,	F,	F,	F,	F]
2	[0,	5,	7,	4,	4,	3,	*]	[–,	0,	0,	2,	2,	2,	–]	[T,	T,	T,	F,	F,	F,	F]
5	[0,	5,	7,	4,	2,	3,	2]	[–,	0,	0,	2,	5,	2,	5]	[T,	T,	T,	F,	F,	T,	F]
4	[0,	5,	7,	4,	2,	3,	2]	[–,	0,	0,	2,	5,	2,	5]	[T,	T,	T,	F,	T,	T,	F]
6	[0,	5,	7,	4,	2,	3,	2]	[–,	0,	0,	2,	5,	2,	5]	[T,	T,	T,	F,	T,	T,	T]
3	[0,	5,	7,	4,	2,	3,	2]	[–,	0,	0,	2,	5,	2,	5]	[T,	T,	T,	T,	T,	T,	T]

'*' indicates infinity.

'–' indicates None.

The first line shows the values of the variables before entering the while-loop (after initialisation).
The remaining lines show the values after completion of each iteration of the while-loop.

6.2 Shortest path trees

In order to find a path that has the shortest distance (sum of weights) from a source vertex to a destination vertex, we solve a more general version of the problem, which is finding the shortest path from the source vertex to every other vertex in the graph. It turns out that the more general version can be solved without loss of efficiency.

Given a single-component weighted undirected graph G, and a vertex s, a shortest-path tree rooted at vertex s is a spanning tree, such that, the path distance from the root s to any other vertex v in the tree is the shortest path distance from s to v in the graph G. Figure 15 shows an example of a shortest path tree. Note that, unlike minimum spanning trees, the shortest path tree is only valid for one source vertex which must be at the root of the tree. For example, in Figure 15, the tree does not provide the shortest path between vertices 4 and 5.

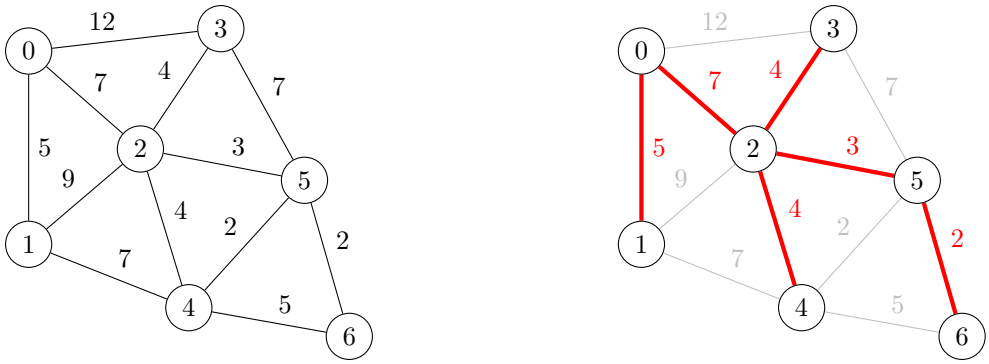


Figure 15: Left: A weighted undirected graph; Right: a shortest path tree rooted at vertex 0. Every path in the tree from vertex 0 to any other vertex, is the shortest distance (smallest sum of weights) among all other possibles paths in the graph from 0 to that vertex.

6.2.1 Dijkstra's algorithm

Given a graph with non-negative edge weights and a starting vertex, the algorithm finds the shortest path from the starting vertex to any other vertex reachable from it.

The algorithm gradually grows a *shortest path tree* rooted at the starting vertex. In each iteration, a new edge is added to the tree by selecting an edge that connects a vertex in the tree to a vertex outside that is closest to the starting vertex.

```
1  procedure DIJKSTRA(Adj, s)
2    n ← number of vertices (the length of Adj)
3    in-tree ← an array of length n filled with FALSE
4    distance ← an array of length n filled with ∞
5    parent ← an array of length n filled with NULL
6    distance[s] ← 0
7    while ¬all(in-tree)
8      u ← NEXT-VERTEX(in-tree, distance)
9      in-tree[u] ← TRUE
10     for v, weight in Adj[u]
11       if ¬ in-tree[v] ∧ distance[u] + weight < distance[v]
12         distance[v] ← distance[u] + weight
13         parent[v] ← u
14   return parent, distance
```

Note the following.

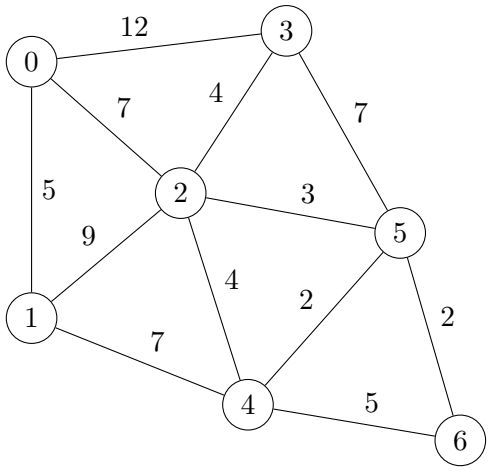
- The algorithm is very similar to Prim's algorithm; only lines 11 and 12 are different because the meaning of distance is different in the two algorithms.
- The *distance* array has the current distance of each vertex from the starting (source) vertex. For vertices that are part of the tree, this is the sum of weights along the shortest path from the starting vertex.
- The *in-tree* array shows the vertices to which the shortest path (from the starting vertex) is known so far.
- Shortest paths can be extracted from the *parent* array.
- The algorithm has the same run-time complexity and Prim's algorithm. It can also be improved in the same way by using an efficient priority queue.

Comments:

- The correctness of Dijkstra's algorithm does not hold for graphs in which some edges have negative weights. For such graphs, the algorithm may produce a non-optimal output (a path that is not the shortest distance).
- An MST and a shortest path tree of a graph are not necessarily the same. In other words, a path from the (arbitrary) root of an MST to another vertex is not necessarily a path with the shortest distance in the graph. Similarly, while a shortest path tree is a spanning tree, it is not necessarily an MST.

Example trace of Dijkstra's algorithm

```
graph_string = """\
U 7 W
0 1 5
0 2 7
0 3 12
1 2 9
2 3 4
1 4 7
2 4 4
2 5 3
3 5 7
4 5 2
4 6 5
5 6 2
"""
```




```
adj_list = adjacency_list(graph_string)

0 [[(1, 5), (2, 7), (3, 12)],
1  [(0, 5), (2, 9), (4, 7)],
2  [(0, 7), (1, 9), (3, 4), (4, 4), (5, 3)],
3  [(0, 12), (2, 4), (5, 7)],
4  [(1, 7), (2, 4), (5, 2), (6, 5)],
5  [(2, 3), (3, 7), (4, 2), (6, 2)],
6  [(4, 5), (5, 2)]]

dijkstra(adj_list, 0)
```

u	Distance of vertex i to root							Parent							In-tree						
	0	1	2	3	4	5	6	0	1	2	3	4	5	6	0	1	2	3	4	5	6
	[0,	*,	*,	*,	*,	*,	*]	[–,	–,	–,	–,	–,	–,	–]	[F,	F,	F,	F,	F,	F,	F]
0	[0,	5,	7,	12,	*,	*,	*]	[–,	0,	0,	0,	–,	–,	–]	[T,	F,	F,	F,	F,	F,	F]
1	[0,	5,	7,	12,	12,	*,	*]	[–,	0,	0,	0,	1,	–,	–]	[T,	T,	F,	F,	F,	F,	F]
2	[0,	5,	7,	11,	11,	10,	*]	[–,	0,	0,	2,	2,	2,	–]	[T,	T,	T,	F,	F,	F,	F]
5	[0,	5,	7,	11,	11,	10,	12]	[–,	0,	0,	2,	2,	2,	5]	[T,	T,	T,	F,	F,	T,	F]
3	[0,	5,	7,	11,	11,	10,	12]	[–,	0,	0,	2,	2,	2,	5]	[T,	T,	T,	T,	F,	T,	F]
4	[0,	5,	7,	11,	11,	10,	12]	[–,	0,	0,	2,	2,	2,	5]	[T,	T,	T,	T,	T,	T,	F]
6	[0,	5,	7,	11,	11,	10,	12]	[–,	0,	0,	2,	2,	2,	5]	[T,	T,	T,	T,	T,	T,	T]

'*' indicates infinity

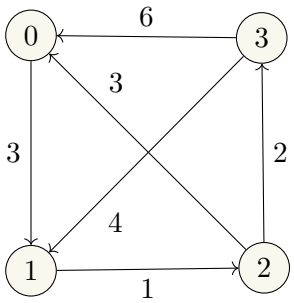
'–' indicates None.

The first line shows the values of the variables before entering the while-loop (after initialisation).
The remaining lines show the values after completion of each iteration of the while-loop.

6.3 All-pairs shortest paths

A single-source shortest path algorithm finds the shortest path from a given source vertex to every vertex in the graph. Sometimes we are interested in the shortest path between every pair of vertices in a graph (i.e. from vertex 0 to every other vertex, from vertex 1 to every other vertex, and so on). One way of achieving all-pairs shortest paths is to run a single-source shortest path algorithm n times, each time starting from a different vertex. Using Dijkstra's algorithm, which has a time complexity of $O(n^2)$, the all-pairs shortest paths problem can be solved in $O(n^3)$ time. Figure 16 shows the result of running such an algorithm on a weighted graph.

```
D 4 W
0 1 3
1 2 1
2 3 2
2 0 3
3 0 6
3 1 4
```



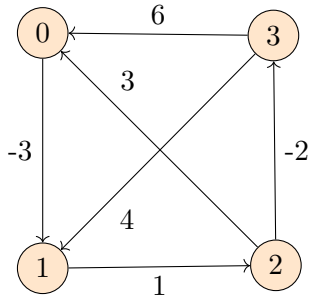
Root	Distance array	Parent array
0	[0, 3, 4, 6]	[-, 0, 1, 2]
1	[4, 0, 1, 3]	[2, -, 1, 2]
2	[3, 6, 0, 2]	[2, 3, -, 2]
3	[6, 4, 5, 0]	[3, 3, 1, -]

Figure 16: Top left: the textual description of a weighted directed graph. Top right: the drawing of the graph. Bottom: the parent and distance arrays obtained by Dijkstra's algorithm. Each distance array shows the shortest distance from the root to each vertex. Each parent array represents the structure of the shortest path tree. The stacking of the distance or parent arrays can be viewed as a square matrix. The element $Distance[i][j]$ is the shortest distance from vertex i to vertex j. The element $Parent[i][j]$ is the parent of vertex j in the shortest path tree that is rooted at vertex i.

One limitation of Dijkstra's algorithm is that, in the presence of negative weights, it may produce non-optimal solutions. This is because the algorithm is greedy; it finds a vertex that is closest to the root and adds it to the shortest path tree. This works fine with non-negative edges because any other path to that vertex has to be at least as long as the first one. With negative edges, however, a greedy algorithm can miss better future options by committing to

a locally optimal choice. Figure 17 shows the result of Dijkstra's algorithm applied to a graph with negative edges. It can be seen that, for some pairs of vertices, the algorithm has found non-optimal paths.

```
D 4 W
0 1 −3
1 2 1
2 3 −2
2 0 3
3 0 6
3 1 4
```



Root	Distance by Dijkstra	Correct distance
0	[0, -3, -2, -4]	[0, -3, -2, -4]
1	[4, 0, 1, -1]	[4, 0, 1, -1]
2	[3, 2, 0, -2]	[3, 0, 0, -2]
3	[6, 4, 5, 0]	[6, 3, 4, 0]

Figure 17: Top left: the textual description of a weighted directed graph that contains some negative weights. Top right: the drawing of the graph. Bottom: the result of applying Dijkstra's algorithm on all the vertices and the true shortest distance between vertices. It can be seen that, in some cases, Dijkstra's algorithm produces non-optimal solutions. For example, starting from vertex 3, Dijkstra sees the edge to 1 with a weight of 4 as the best choice, while a worse local choice (the edge from 3 to 0 with a weight of 6) can later lead to a shorter path to vertex 1.

6.3.1 Developing a recursive solution

In this section, we develop a recursive function for the shortest path problem which will lead to a DP solution for the all-pairs shortest paths problem, called the Floyd-Warshall algorithm. While we will address the problem of producing optional solutions for graphs with edges that have negative weights, we limit the solution to graphs that do not have *negative cycles* which are defined as cycles along which the sum of weights is a negative value.

6.3.1.1 Intermediate vertices

Consider a path $p = (v_s, v_1, v_2, ..., v_l, v_d)$ which is from the source vertex v_s to the destination vertex v_d. We call the vertices in between the source and destination *intermediate vertices*. The path p has l intermediate vertices denoted by v_1, v_2, ..., v_l.

6.3.1.2 Properties of shortest paths

The following properties hold for shortest paths:

- Shortest paths are simple paths; that is, no vertex is repeated along the path. Why? Because repeating a vertex (in the absence of a negative cycle) cannot make the sum of the weights smaller.
- Any subpath (consecutive subsequence of vertices) of a shortest path is also a shortest path. Why? If this was not the case, then there would be a shorter path connecting the vertex at the beginning of the subpath to the vertex at the end of the subpath. This would mean that the original path is not the shortest path, which is a contradiction.

6.3.1.3 A new distance function

Consider a graph with n vertices identified by numbers from 0 to $n-1$. We define the function $d(i, j, k)$ to be the shortest distance from vertex i to vertex j if we only consider paths whose intermediate vertices are in $\{0, 1, \ldots, k\}$. For example, $d(i, j, 1)$ is the shortest distance from i to j if we only consider the following paths (if they exit):

- a path with no intermediate vertex, that is, an edge from i to j;
- a path from i to 0 and then to j;
- a path from i to 1 and then to j;
- a path from i to 0, to 1, and then to j; and
- a path from i to 1, to 0, and then to j.

The parameters i and j are integers between 0 and $n-1$ corresponding to the vertices of the graph. The parameter k can take values between -1 and $n-1$. The value of -1 means that no intermediate vertices can be used. The value of $n-1$ means that intermediate vertices can be any vertex in the graph (i.e. all paths are considered). Figure 18 shows a few examples of the value of this function on a graph.

i	j	k	intermediates allowed	distance	path
3	1	-1	$\{\}$	4	$(3, 1)$
2	1	-1	$\{\}$	∞	-
3	1	0	$\{0\}$	3	$(3, 0, 1)$
2	1	0	$\{0\}$	0	$(2, 0, 1)$
3	2	0	$\{0\}$	∞	-
3	2	1	$\{0, 1\}$	4	$(3, 0, 1, 2)$
1	0	1	$\{0, 1\}$	∞	-
1	0	2	$\{0, 1, 2\}$	4	$(1, 2, 0)$
1	0	3	$\{0, 1, 2, 3\}$	4	$(1, 2, 0)$

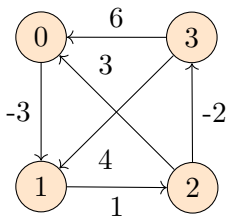


Figure 18: The table on the left presents some examples of the value of the function $d(i, j, k)$ for the graph on the right.

The value of $d(, , k)$ can be recursively expressed based on the values of the form $d(, , k-1)$. When evaluating $d(i, j, k)$, two cases can arise:

- k is an intermediate vertex in the shortest path from i to j. In this case, the path can be seen as the concatenation of the shortest path from i to k and the shortest path from k to j. The intermediate vertices of these two paths are in $\{0, 1, \ldots, k-1\}$. Note that k is not an intermediate vertex, because in one of the paths, it is the destination, and in the other one, it is the source. Therefore, the lengths of these two paths are $d(i, k, k-1)$ and $d(k, j, k-1)$ respectively.
- k is not an intermediate vertex in the shortest path from i to j. In this case, the intermediate vertices are in $\{0, 1, \ldots, k-1\}$. Therefore, the length of the path is the same as $d(i, j, k-1)$.

This leads to the following recurrence equation:

$$d(i,j,k) = \begin{cases} 0, & \text{if } k = -1 \wedge i = j \\ \infty, & \text{if } k = -1 \wedge (i,j) \notin E \\ w((i,j)), & \text{if } k = -1 \wedge (i,j) \in E \\ \min(d(i,k,k-1) + d(k,j,k-1), d(i,j,k-1)), & \text{if } k \geq 0 \ . \end{cases}$$

In this equation:

- The first three cases are the base cases, where $k = -1$; that is, no intermediate vertex is used.
- The first base case is when the source and destination are the same.
- The second base case is when there is no edge directly connecting i to j and no intermediate vertices are allowed. In this case the distance is defined to be infinity.

- The third case is when there is a direct edge connecting i to j. In this case, the distance is the weight of the edge.
- The last case is the recursive case. The shortest distance from i to j is the smaller of the two cases mentioned above: one has k as the intermediate vertex, the other does not.

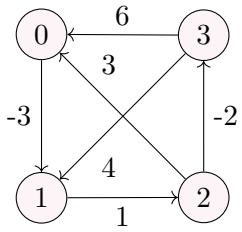
6.3.2 The Floyd-Warshall algorithm

The Floyd-Warshall algorithm takes a weighted (directed or undirected) graph with no negative cycle and computes the shortest distance between every pair of vertices.

The algorithm can be implemented recursively according to the recursive definition of $d(i, j, k)$. Since there will be repeated recursive calls, memoisation must be used. This leads to a top-down DP solution. Since each of the parameters i, j, and k can take on n distinct values, the size of the cache will be in $O(n^3)$. The body of the function itself is very simple and runs in constant time. Thus the running time of the algorithm is proportional to the number of unique recursive calls which is in $O(n^3)$.

6.3.2.1 Bottom-up implementation

A bottom-up version of the algorithm can be implemented by starting from a distance matrix which contains the values of $d(i, j, k)$ for $k = -1$. Figure 19 shows an example of such a matrix. Then the matrix can be updated iteratively by considering more intermediate vertices in each iteration. This leads to the following algorithm.




```
          j
i |    0     1     2     3
--------------------------
0 |    0,   -3,  inf,  inf
1 |  inf,    0,    1,  inf
2 |    3,  inf,    0,   -2
3 |    6,    4,  inf,    0
```

Figure 19: Left: a weighted directed graph. Right: the values of the function $d(i, j, -1)$ which represents the shortest distance from vertex i to vertex j when no intermediate vertex is used. The matrix on the right is similar to an adjacency matrix. The only differences are: (a) when there is no edge from i to j, the value is set to infinity (instead of NULL); and (b) when $i = j$, the distance is considered to be zero. This matrix is the starting state of the (bottom-up) Floyd-Warshall algorithm.

```
1  procedure FLOYD(Distance)
2    n ← number of rows (or columns) of the matrix Distance
3    for k from 0 to n − 1:
4      for i from 0 to n − 1:
5        for j from 0 to n − 1:
6          if Distance[i][j] > Distance[i][k] + Distance[k][j]
7            Distance[i][j] ← Distance[i][k] + Distance[k][j]
8    return Distance
```

In the algorithm:

- The matrix *Distance* initially contains all the single edge (without intermediate vertices) distances. This information is directly available from the graph.
- In each iteration of the outer loop, a new vertex k is considered as an intermediate vertex in computing the shortest path between every pair of vertices.
- At the end of the algorithm, $Distance[i][j]$ has the shortest path from i to j.
- The runtime complexity of the algorithm is in $\Theta(n^3)$.
- The algorithm takes advantage of the fact that the value of $d(i, j, k)$ only depends on the values of $d(i, k, k-1)$ and $d(k, j, k-1)$. Thus, it only keeps one distance matrix in memory and incrementally updates the matrix. This means that the space complexity of the algorithm is in $\Theta(n^2)$.
- The algorithm can be extended to update a parent matrix. At the beginning, $parent[i][j]$ is set to i if there is an edge (i, j) in the graph. Later, every time the distance matrix is updated, $parent[i][j]$ is set to $parent[k][j]$.

Trace of Floyd-Warshall algorithm

initial values (k=-1)

distance

	0	1	2	3
0	0,	-3,	inf,	inf
1	inf,	0,	1,	inf
2	3,	inf,	0,	-2
3	6,	4,	inf,	0

parent

	0	1	2	3
0	None,	0,	None,	None
1	None,	None,	1,	None
2	2,	None,	None,	2
3	3,	3,	None,	None

k = 0

distance

	0	1	2	3
0	0,	-3,	inf,	inf
1	inf,	0,	1,	inf
2	3,	0,	0,	-2
3	6,	3,	inf,	0

parent

	0	1	2	3
0	None,	0,	None,	None
1	None,	None,	1,	None
2	2,	0,	None,	2
3	3,	0,	None,	None

k = 1

distance

	0	1	2	3
0	0,	-3,	-2,	inf
1	inf,	0,	1,	inf
2	3,	0,	0,	-2
3	6,	3,	4,	0

parent

	0	1	2	3
0	None,	0,	1,	None
1	None,	None,	1,	None
2	2,	0,	None,	2
3	3,	0,	1,	None

k = 2

distance

	0	1	2	3
0	0,	-3,	-2,	-4
1	4,	0,	1,	-1
2	3,	0,	0,	-2
3	6,	3,	4,	0

parent

	0	1	2	3
0	None,	0,	1,	2
1	2,	None,	1,	2
2	2,	0,	None,	2
3	3,	0,	1,	None

k = 3

distance

	0	1	2	3
0	0,	-3,	-2,	-4
1	4,	0,	1,	-1
2	3,	0,	0,	-2
3	6,	3,	4,	0

parent

	0	1	2	3
0	None,	0,	1,	2
1	2,	None,	1,	2
2	2,	0,	None,	2
3	3,	0,	1,	None

* Values have been printed after the completion of each iteration of the outer loop.

7 Backtracking

Backtracking is an algorithmic technique for generating all or some solutions of a computational problem. Examples include:

- generating permutations of a set of items;
- generating subsets of a set;
- completing a Sudoku puzzle; and
- placing 8 queens on a chess board so that they do not attack each other.

The idea is that candidate solutions can be constructed incrementally.

Example 7.1. When constructing a string representing a binary number of certain length, one can start with an empty string. Then, recursively, the next candidates are constructed by appending 0 or 1 to a copy of a candidate from the previous stage. This process is repeated until the desired length is reached.

The following program prints all binary numbers of a given length.

```
def print_binary_numbers(desired_length, string=""):
    if len(string) == desired_length:
        print(string)
    else:
        print_binary_numbers(desired_length, string + "0")
        print_binary_numbers(desired_length, string + "1")
```

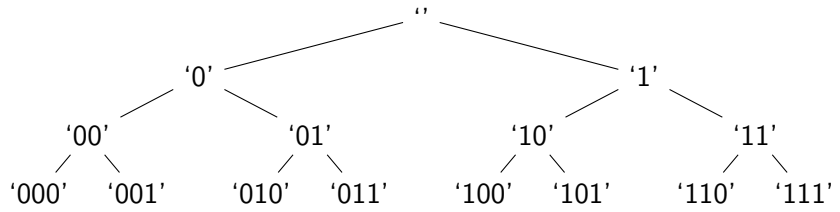
Sample output:

```
> print_binary_numbers(3)

000
001
010
011
100
101
110
111
```

The incremental generation of candidates can be seen as a tree. There is a vertex for each candidate. There is an edge from a candidate to another candidate if and only if the latter can be constructed by adding an element to the former.

Example 7.2. Consider the problem of generating all binary numbers of length n. Draw the tree of candidates for $n = 3$.



In most problems these trees grow very rapidly with respect to the size of the problem.

7.1 Backtracking algorithm

A traversal (BFS or DFS) can be used to search the tree for solutions.

Typically there are regularities in the graph of candidates. Each problem has its own recursive pattern. Therefore, the graph can be implicitly constructed on the fly. Since the graph is a tree, given its root it can be fully described with a CHILDREN function.

The backtracking algorithm is a DFS on this implicit tree.

```
1  procedure DFS-BACKTRACK(candidate, input, output)
2    if IS-SOLUTION(candidate, input)
3      ADDTOOUTPUT(candidate, output)
4    else
5      for child-candidate in CHILDREN(candidate, input)
6        DFS-BACKTRACK(child-candidate, input, output)
```

Elements of the algorithm are:

- *candidate* is a vertex of the implicit tree.
- *input* contains information about the instance of the problem. For example, when generating binary numbers, the input is the desired length.
- *output* is where the solutions are collected. For example, it can be a list that at the end of the algorithm will contain all binary numbers.

- Is-Solution is a predicate (a Boolean-valued function) that returns true if the candidate is a solution (a leaf vertex). For example, when generating binary strings, it returns true if the candidate has the desired length.
- Children is a function that takes a candidate and returns a (possibly empty) sequence of candidates that can be constructed by augmenting the given candidate. Augmentation usually involves adding another element to a copy of the given candidate. The function Children often needs to refer to the *input* in order to construct the new candidates. For example, when generating binary numbers of a certain length, the children of a candidate are candidates constructed by appending 0 or 1 to the end of the given candidate binary string.

7.1.1 Example implementation

```
def dfs_backtrack(candidate, input, output):
    if is_solution(candidate, input):
        add_to_output(candidate, output)
    else:
        for child_candidate in children(candidate):
            dfs_backtrack(child_candidate, input, output)


def is_solution(candidate, desired_length):
    return len(candidate) == desired_length


def children(candidate):
    return [candidate + "0", candidate + "1"]


def add_to_output(candidate, output):
    output.append(candidate)


def binary_numbers(desired_length):
    solutions = []
    dfs_backtrack("", desired_length, solutions)
    return solutions
```

7.1.2 Sample output

```
> binary_numbers(4)

['0000',
 '0001',
 '0010',
 '0011',
 '0100',
 '0101',
 '0110',
 '0111',
 '1000',
 '1001',
 '1010',
 '1011',
 '1100',
 '1101',
 '1110',
 '1111']
```

7.2 Backtracking for search

Backtracking can be used to search for a solution. For example in Sudoku the algorithm tries various placements of numbers in empty cells until it finds a solution (where all cells are filled and all criteria of Sudoku are met). If only one solution is desired, the algorithm can be modified to set a flag as soon as it finds a solution so that the search can be stopped.

The algorithm is called backtracking because it makes some choices when branching down but it can "backtrack" by returning from recursive calls and try other branches.

7.2.1 Pruning

The performance of the algorithm can be improved by *pruning*, which means some nodes are not expanded because they cannot lead to solution. For example, when searching for a solution of a Sudoku problem, if the candidate already breaches some of the criteria for being a valid solution, there is no point in expanding it any further. At this point the (recursive) call can return.

To implement pruning the following statement is added to the beginning of DFS-Backtrack:

if Should-Prune(*candidate*)
    return

The function Should-Prune takes a candidate and returns true if it can be determined that no leaf candidate in the subtree rooted at the given candidate is a solution.

For example, when searching for binary numbers of a given length, if we are only interested in binary numbers that have up to four '1's in them, the function can be implemented as the following:

```
def should_prune(candidate):
    return candidate.count("1") > 4
```

8 Linear Time Sorting

So far you have seen several sorting algorithms. Some of these algorithms have an upper bound of $O(n^2)$ for their running time (for example, insertion sort and selection sort) and some have an upper bound of $O(n \log n)$ (for example, heap sort, merge sort and quick sort).

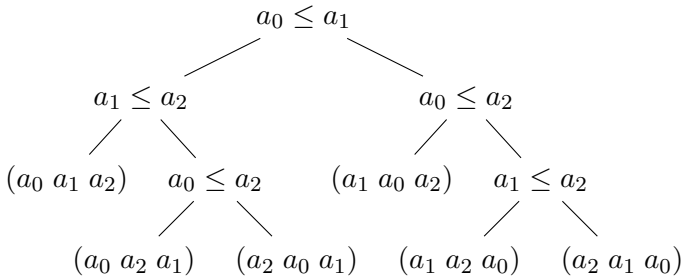
These algorithms share a common property: they *compare* the key of items to be sorted. For this reason we call them **comparison sort** algorithms.

8.1 Lower bound for comparison sorts

In comparison sorting, the algorithm gains information about the order of items by comparing input elements two at a time. For example, for an input sequence of $a_0, a_1, \ldots, a_n$, the algorithm makes comparisons of the form $a_i \leq a_j$ to determine which one comes earlier.

The collection of all the comparisons required for an algorithm to determine the correct order of elements can be represented as a full binary decision tree where the internal (non-leaf) nodes are comparisons and the leaf nodes are possible orderings.

Example 8.1. Draw the comparison decision tree of the sequence $[a_0, a_1, a_2]$.



Theorem 8.1. *Any comparison sort algorithm requires* $\Omega(n \log n)$ *comparisons in the worst case.*

Proof. Assume n distinct elements:

- number of possible outputs:
- minimum number of leaf nodes:
- minimum height of the tree:
- worst-case time (maximum path length):

8.2 How about non-comparison sorting?

The lower bound in the above theorem applies to problems where the key values must be compared. If the key function returns a natural number from a bounded range, better running times can be achieved!

For the following it helps to distinguish between the objects in the input and their key values, so comparisons take the form $key(a_i) \leq key(a_j)$.

Question 8.1. You have an array $A[0\,..\,n]$ of objects. You are to sort the array based on a key whose value is between 0 and n and is guaranteed to be distinct (that is, no two objects will have the same key value). Give a linear time algorithm to achieve this.

Example:

$object$	a_0	a_1	a_2	a_3	a_4	a_5
$key(object)$	2	4	3	1	5	0

Question 8.2. You have an array $A[0\,..\,n]$ of objects. You are to sort the array based on a key whose value is between 0 and k and is guaranteed to be distinct and we have $k \geq n$. Give a linear time algorithm to achieve this.

Example:

$object$	a_0	a_1	a_2	a_3	a_4
$key(object)$	0	3	6	4	1

Question 8.3. You have an array $A[0\,..\,n]$ of objects. You are to sort the array based on a key whose value is between 0 and k with $k \geq n$. Give a linear time algorithm to achieve this.

Example:

$object$	a_0	a_1	a_2	a_3	a_4
$key(object)$	6	1	6	1	1

8.3 Counting sort

Counting sort assumes that each of the n input elements has a key value in the range 0 to k, for some natural number k. The algorithm has two stages:

1. The algorithm calls the procedure KEY-POSITIONS which counts the number of times each key value occurs in the input array and uses this information to compute the position of objects with that key in the output (sorted) array. It returns an array of positions of length $k + 1$.

2. The algorithm uses the positions obtained in the first stage to place the elements of the input at the right position in the output array.

```
1  procedure COUNTING-SORT(A, key)
2    create an output array B with the same size as A
3    P ← KEY-POSITIONS(A, key)
4    for a in A
5      B[P[key(a)]] ← a
6      P[key(a)] ← P[key(a)] + 1
7    return B
```

The input to the algorithm is an array A of objects and a key function according to which a sorted output array must be generated. After line 3, $P[i]$ contains the starting index for objects whose key value is i. In other words, the first object in A whose key value is i must be placed at index $P[i]$ in the output array. The for-loop goes through the elements of A and places each element in its correct sorted position in the output array B, and updates P.

```
1  procedure KEY-POSITIONS(A, key)
2    k ← max{key(a) : a ∈ A}
3    let C[0..k] be a new array
4    for i from 0 to k
5      C[i] ← 0
6    for a in A
7      C[key(a)] ← C[key(a)] + 1
8    sum ← 0
9    for i from 0 to k
10     C[i], sum ← sum, sum +C[i]
11   return C
```

At line 2 the algorithm finds the maximum value of the key function over the input array A and stores it in k. At lines 3–5, a new array C of size $k + 1$ is created for keeping counts; it is initialised with zeros.

After the for-loop at lines 6–7, $C[i]$ is the number of elements whose key value is i. The for-loop at lines 9–10 computes a running sum over C. At the end of the for-loop, $C[i]$ is the number of elements whose key value is less than i.

Example 8.2. Trace KEY-POSITIONS and COUNTING-SORT on the following input.

object	a_0	a_1	a_2	a_3	a_4	a_5	a_6	a_7
key(*object*)	2	5	3	0	2	3	0	3

Value of C during lines 6 and 7:

Value of C during lines 9 and 10:

Values of B and P during the execution of COUNTING-SORT.

8.3.1 Analysis and properties of counting sort

Let n be the size of the input array. The time complexity of KEY-POSITIONS is as follows. Line 2 takes $\Theta(n)$ time. The three for-loops take $\Theta(k)$, $\Theta(n)$, and $\Theta(k)$ time respectively.

In the main body of the counting sort algorithm, the for-loop takes $\Theta(n)$ time. Therefore the time complexity of the counting sort algorithm is $\Theta(k+n) = \Theta(\max(k,n))$. This means that if $k = O(n)$, the algorithm runs in $\Theta(n)$ time.

An important property of counting sort is that it is **stable**: objects with the same key value appear in the output array in the same order as they do in the input array. This is equivalent to saying that the algorithm breaks ties between two objects with the same key value by the rule that whichever object appears first in the input array appears first in the output array.

Example 8.3. Let $A = [4, 2, 4, 3, 2, 3]$ and let the key be the identity function. What is the intermediate state and output of KEY-POSITIONS on this input?

The array containing key value counts is: $[0, 0, 2, 2, 2]$.
The output is $[0, 0, 0, 2, 4]$.

Example 8.4. Let $A = [a, b, c, d, e, f, g, h]$ where each element is an object and the key values are:

object	a	b	c	d	e	f	g	h
$key(object)$	2	4	5	0	2	4	3	5

1. What is the output of KEY-POSITIONS? Show one intermediate step.
2. Trace the procedure COUNTING-SORT using the key positions from the previous step.

1. The key counts (in the intermediate step) are $[1, 0, 2, 1, 2, 2]$. The output of KEY-POSITIONS is $[0, 1, 1, 3, 4, 6]$.
2.

P	*Sorted array*
(0, 1, 1, 3, 4, 6)	(-, -, -, -, -, -, -, -)
(0, 1, 2, 3, 4, 6)	(-, a, -, -, -, -, -, -)
(0, 1, 2, 3, 5, 6)	(-, a, -, -, b, -, -, -)
(0, 1, 2, 3, 5, 7)	(-, a, -, -, b, -, c, -)
(1, 1, 2, 3, 5, 7)	(d, a, -, -, b, -, c, -)
(1, 1, 3, 3, 5, 7)	(d, a, e, -, b, -, c, -)
(1, 1, 3, 3, 6, 7)	(d, a, e, -, b, f, c, -)
(1, 1, 3, 4, 6, 7)	(d, a, e, g, b, f, c, -)
(1, 1, 3, 4, 6, 8)	(d, a, e, g, b, f, c, h)

8.4 Radix sort

In counting sort if k becomes larger than n, the algorithm loses its linear time property. So how to sort an array of large numbers?

The idea of radix sort is to sort numbers digit by digit. Counterintuitively, radix sort starts sorting numbers first based on their *least significant* (rightmost) digit, then sorts the result based on the second-least significant digit, and so on. If the numbers in the input array have at most d digits, then after d iterations the resulting array is sorted.

```
1 procedure RADIX-SORT(A, d)
2   for i from 1 to d
3     use a stable sort to sort array A on digit i
```

Radix sort requires another sorting algorithm that is stable. Counting sort is usually used for this purpose.

Example 8.5. Let $A = [329, 457, 657, 839, 436, 720, 355]$. Sort the array using radix sort. Show the result of each iteration. ...

Example 8.6. Let $A = [543, 774, 166, 298, 7, 54, 12]$. Trace the procedure call RADIX-SORT$(A, 2)$.

Iteration 1: $[12, 543, 774, 54, 166, 7, 298]$
Iteration 2: $[7, 12, 543, 54, 166, 774, 298]$.

9 Reductions and hardness of problems

Reduction is the act of *reducing* one problem to another. Problem X can be reduced to problem Y if and only if every instance of X can be turned into an instance of Y and the solution of this instance of Y can be converted back into a solution for X.

Example 9.1. The problem of finding the minimum of an array of numbers can be reduced to the problem of finding the maximum element in the array.

```
def find_min_using_max(numbers):
    negated_numbers = [-x for x in numbers]
    maximum = max(negated_numbers)
    minimum = -maximum
    return minimum
```

Answer the following questions:

- What is the time complexity of this algorithm?
- What can be said about the difficulty of the problem of finding the min?
- What can be said about the difficulty of the problem of finding the max, if we know that the time complexity of *any* algorithm for finding the min is in $\Omega(n)$?

More generally, a computational problem X can be reduced to a computational problem Y if there is a procedure such as the following:

```
1 procedure SOLVE-X-USING-Y(X-instance)
2   Y-instance ← CONVERT-INSTANCE-X-TO-Y(X-instance)
3   Y-solution ← SOLVE-Y(Y-instance)
4   X-solution ← CONVERT-SOLUTION-Y-TO-X(Y-solution)
5   return X-solution
```

Reduction of X to Y allows two types of conclusions:

- if Y is easy, then X is easy.
- if X is hard, then Y is hard.

Y is said to be easy if an efficient algorithm for solving it is known.

X is said to be hard if it has been proved that no efficient algorithm for solving X can exist or no efficient algorithm for solving X is known.

More precisely,

- if the time complexity of conversion (of both instance and solution) is in $O(f_{convert})$, and the time complexity of solving Y is in $O(f_{solveY})$, then the time complexity of X is in $O(f_{solveY} + f_{convert})$;
- if we know that any algorithm for X is in $\Omega(f_{solveX})$, then any algorithm for solving Y is in $\Omega(f_{solveX} - f_{convert})$.

Example 9.2. The problem of finding the minimum of an array of numbers can be reduced to the problem of sorting the array.

```
def find_min_using_sort(list_of_numbers):
    sorted_list = comparison_sort(list_of_numbers)
    return sorted_list[0]
```

Answer the following questions:

- What can be said about the worst-case running time of this function without making any assumption about the sorting algorithm (other than it is a comparison sort)? [The two rules mentioned above are not needed to answer this question.]
- If we can prove that all algorithms for finding min are in $\Omega(n)$, what can be said about the problem of sorting?
- Knowing that there are sorting algorithms that run in $O(n \log n)$, what can be said about the problem of finding the min.
- If we devise a separate algorithm that can solve the problem of finding the min in $O(n)$, what can be said about the sorting problem (from the above reduction)?
- Considering the fact that the worst-case running time of a comparison sorting algorithm is in $\Omega(n \log n)$, what can be said about the difficulty of the min problem?

Example 9.3. Reduce the problem of sorting an array of numbers to the topological sorting problem. What claims can be made for the complexity of the two problems?