# Design and Analysis of Algorithms

## Contents


**1 Analysis** **1**
- 1.1 Asymptotic Notation . . . 1
  - 1.1.1 Big-O Notation . . . 1
  - 1.1.2 Big-Omega Notation . . . 2
  - 1.1.3 Big-Theta Notation . . . 2
- 1.2 Recurrence Relations . . . 2
  - 1.2.1 The Master Theorem . . . 2
- 1.3 Correctness and Invariants . . . 2

**2 Design Principles** **2**
- 2.1 Divide and Conquer . . . 2
  - 2.1.1 Merge Sort . . . 2
  - 2.1.2 Binary Search . . . 3
  - 2.1.3 Selection Problem . . . 3
  - 2.1.4 Integer Multiplication . . . 3
  - 2.1.5 Matrix Multiplication . . . 3
- 2.2 Dynamic Programming . . . 4
  - 2.2.1 Knapsack Problems with Repetition . . . 4
  - 2.2.2 Knapsack Problems without Repetition 4
  - 2.2.3 Change Making Problem . . . 4
  - 2.2.4 Edit Distance Problem (Levenshtein Distance) . . . 4
  - 2.2.5 Travelling Salesman Problem . . . 5
  - 2.2.6 All-pairs shortest path . . . 5
- 2.3 Greedy Algorithms . . . 5
  - 2.3.1 Change Making Algorithms . . . 5
  - 2.3.2 Minimum Spanning Tree Algorithm . . . 5
  - 2.3.3 Kruskal's Algorithm . . . 6
  - 2.3.4 Prim's Algorithm . . . 6
- 2.4 Dynamic Programming vs. Divide-and-Conquer 6

**3 Data Structures** **6**
- 3.1 Heaps . . . 6
  - 3.1.1 Representation . . . 6
  - 3.1.2 Maintaining heaps . . . 6
  - 3.1.3 Heap-Sort . . . 7
- 3.2 Priority Queues . . . 7
  - 3.2.1 Implementation . . . 7
- 3.3 Queues . . . 7
  - 3.3.1 Implementation . . . 8
- 3.4 Graphs . . . 8
  - 3.4.1 Definitions . . . 8
  - 3.4.2 Representations . . . 8
  - 3.4.3 Depth-first Search (DFS) . . . 8
  - 3.4.4 Topological Sort . . . 9
  - 3.4.5 Strongly Connected Components . . . 9
  - 3.4.6 Breadth-First Search (BFS) . . . 9
  - 3.4.7 Shortest Path Problems . . . 9
  - 3.4.8 Djikstra's Algorithm . . . 9
  - 3.4.9 Floyd-Warshall . . . 10
  - 3.4.10 Bellman-Ford . . . 10


# List of Algorithms


1 Merge Sort . . . 2
2 Binary Search . . . 3
3 Partition algorithm . . . 3
4 Selection Algorithm . . . 3
5 Knapsack Problem with repetition . . . 4
6 Knapsack Problem without repetition . . . 4
7 Change giving algorithm . . . 5
8 Levenshtein Distance . . . 5
9 Floyd-Warshall Algorithm . . . 5
10 Generic MST algorithm . . . 5
11 Kruskal's Algorithm . . . 6
12 Prim's Algorithm . . . 6
13 Heapify algorithm . . . 7
14 Make-Heap algorithm . . . 7
15 Heap-Sort . . . 7
16 Return the maximal element of the queue . . . 7
17 Extract Max Algorithm . . . 7
18 Increase Key Algorithm . . . 8
19 Insertion Algorithm . . . 8
20 Depth-First Search Algorithm . . . 8
21 Topological Sort . . . 9
22 Strongly Connected Component Discovery . . . 9
23 Bread-First Search . . . 9
24 Djikstra's Shortest Path Algorithm . . . 10
25 Bellman-Ford Algorithm . . . 10


## 1 Analysis

### 1.1 Asymptotic Notation

#### 1.1.1 Big-O Notation

For the functions $f, g : \mathbb{N} \mapsto \mathbb{R}_{\geq 0}$, let the set

$$O(g(n)) := \{f : \mathbb{N} \mapsto \mathbb{R}^+ : \exists n_0 \in \mathbb{N}^+ \cdot \exists c \in \mathbb{R}^+ \cdot \\ n \geq n_0 \rightarrow f(n) \leq c \times g(n)\}$$

$f \in O(g)$, if, for some large enough $n$, $f(n) \leq c \times g(n)$ for some constant factor $c$. The set $O(g)$ is the set of functions bounded above by $g$, without taking into account constant factors.

When the equals sign is used in Big-O notation, it does not imply the usual property of symmetry associated with equality. $n = O(n^2)$, but $n^2 \neq O(n)$. Additionally, $n = O(n^3)$ and $n^2 = O(n^3)$, but $n \neq n^2$.

Big-O notation has the following properties:

- $\forall c \in R_{\geq 0}$, if $f \in O(g)$ then $cf \in O(g)$
- $\forall c \in R_{\geq 0}$, if $f \in O(g)$ then $f \in O(cg)$, (hence where writing log operations without bases comes from).
- If $f_1 \in O(g_1)$ and $f_2 \in O(g_2)$ then $f_1 + f_2 \in O(g_1 + g_2)$
- If $f_1 \in O(g_1)$ and $f_2 \in O(g_2)$ then $f_1 + f_2 \in O(\max(g_1, g_2))$
- If $f_1 \in O(g_1)$ and $f_2 \in O(g_2)$ then $f_1 \times f_2 \in O(g_1 \times g_2)$
- If $f \in O(g)$, and $g \in O(h)$, then $f \in O(h)$
- Every polynomial of degree $l \geq 0$ is in $O\left(n^l\right)$
- $\forall c \in \mathbb{R} \cdot \lg(n^c) \in O(\lg n)$
- $\forall c, d > 0 \cdot \lg^c n \in O\left(n^d\right)$
- $\forall c > 0 \wedge d > 0 \cdot n^c \in O(d^n)$

### 1.1.2 Big-Omega Notation

For the functions $f, g : \mathbb{N} \mapsto \mathbb{R}_{\geq 0}$, let the set

$$\Omega(g(n)) := \{f : \mathbb{N} \mapsto \mathbb{R}^+ : \exists n_0 \in \mathbb{N}^+ \cdot \exists c \in \mathbb{R}^+ \cdot \\ n \geq n_0 \rightarrow f(n) \geq c \times g(n)\}$$

$f \in \Omega(g)$, if, for some large enough $n$, $f(n) \geq c \times g(n)$ for some constant factor $c$. The set $O(g)$ is the set of functions bounded below by $g$, without taking into account constant factors.

### 1.1.3 Big-Theta Notation

For the functions $f, g : \mathbb{N} \mapsto \mathbb{R}_{\geq 0}$, if $f \in \Theta(g)$, then $g$ is an asymptotic upper and lower bound to $f$. That is, for $c_1, c_2 \in \mathbb{R}_{>0}$ and $n_0$ such that, for all $n \geq n_0$

$$c_1 g(n) \leq f(n) \leq c_2 g(n)$$

## 1.2 Recurrence Relations

There are four methods to solve recurrence relations:

1. Recursion tree – by unfolding at each level, and substituting in the non-recursive term, and approximation for each level can be found. By determining the number of levels, and the cost of the final level, a total cost can be discovered.
2. Iterative substitution – similar to above, but without drawing the diagram.
3. Guess-and-Test – by substituting in an approximation, either a contradiction will be discovered $-T(n) \geq 0$, or none will be found, hence the correct form of $T(n)$ has been discovered.
4. The Master Theorem

### 1.2.1 The Master Theorem

Suppose

$$T(n) = \begin{cases} O(1) & \text{if } n = 1 \\ aT(\lceil n/b \rceil) + O(n^d) & \text{if } n > 1 \end{cases}$$

then for some constants $a > 0, b > 1, d \geq 0$

$$T(n) = \begin{cases} O(n^d) & \text{if } d > \log_b a \\ O(n^d \log_b n) & \text{if } d = \log_b a \\ O(n^{\log_b a}) & \text{if } d < \log_b a \end{cases}$$

## 1.3 Correctness and Invariants

Loop-Invariant correctness proofs allow a program to be proven to be correct by a process similar to mathematical induction.

**Initialisation** – prove that an invariant $I$ holds just before the first iteration (base case)

**Maintenance** – prove that if $I$ holds just before the start of a loop, it holds just before the next loop (inductive case)

**Termination** – prove that, when the loop terminates, $I$ and the reason for termination guarantee the end result is correct.

# 2 Design Principles

## 2.1 Divide and Conquer

The divide-and-conquer strategy can be thought of as solving problems in the following steps:

1. Break the initial problem into subproblems.
2. Recursively solve the subproblems [if the problems are small enough, solve by brute force for a *base case*].
3. Appropriately combine the answers to the subproblems.

The most complicate work is found in dividing the problems into the subproblems, at the tail-end of the recursion, when solving the subproblems, and gluing the intermediate answers together.

### 2.1.1 Merge Sort

**Description** This is a divide-and-conquer algorithm to sort the array $A[p..r)$.

1. *Divide* – split the array into $A[p..q)$ and $A[q..r)$, where $q = \lfloor (p+q)/2 \rfloor$.
2. *Conquer* – by recursively sorting the subarrays, and bottoming out the recursion when singleton arrays are reached.
3. *Combine* – by merging the sorted subarrays $A[p..q)$ and $A[q..r)$ using a $\Theta(n)$ procedure.

**Algorithm 1:** Merge Sort

```
1  def MERGE-SORT (A, p, r)
       Input: An integer array A with indices p < r
       Output: The
                 subarray A[p..r) sorted in increasing order
2      if p + 1 < r then
3          q = ⌊(p + r)/2⌋;
4          MERGE-SORT (A, p, q);
5          MERGE-SORT (A, q, r);
6          MERGE (A, p, q, r);

1  def MERGE (A, p, q, r)
       Input: Array
                A with indices p, q, r such that p < q < r and
                subarrays A[p..q) and A[q..r) already sorted.
       Output: The subarrays
                 are merged into a sorted array A[p..r)
2      n1 = q − p, n2 = r − q;
3      Create array L of size n1+1, Create array R of size n2+1;
4      for i = 0 to n1 − 1 do L[i] = A[p + i];
5      for j = 0 to n2 − 1 do R[j] = A[q + j];
6      L[n1] = ∞, R[n2] = ∞;
7      i = 0, j = 0;
8      for k = p to r − 1 do
9          if L[i] ≤ R[j]/* To ensure stable sort    */
10         then A[k] = L[i];i = i + 1;
11         else A[k] = R[j];j = j + 1;
12     return A;
```

**Pseudo-code** `MERGE` runs in $\Theta(n)$ as each of the `for` loops' lines run in constant time, iterating over $n$ elements, therefore the entire function is bounded by $n$.

**Advantages**

- Merge sort is stable, as the `MERGE` function is left biased.
- Runtime is always $O(n \log n)$.

**Disadvantages**

- Merge, and by extension, merge sort requires $O(n)$ extra space.
- Merge sort is not an online algorithm.

**Haskell Implementation** By pattern matching, first against the base cases, the program then recurses into two more calls of mergesort using two separate halves of the list. The call to merge then ensure that the two halves of the list are joined into a correctly sorted list.

```
mergesort :: Ord a ⇒ [a] → [a]
mergesort [] = []
mergesort [x] = [x]
mergesort xs = merge (mergesort xs1) (mergesort xs2)
        where (xs1,xs2) = split xs

split :: [a] → ([a],[a])
split (x:y:zs) = (x:xs,y:ys) where (xs,ys) = split zs
```

```
split xs = (xs,[])

merge :: [a] → [a] → [a]
merge xs [] = xs
merge [] ys = ys
merge (x:xs) (y:ys) =
  case x ≤ y of
       True → x : merge xs (y:ys)
       False → y : merge (x:xs) ys
```

### 2.1.2 Binary Search

**Description and Requirements**

- The array to work with is sorted
- Begin by looking at the middle element. If the item to be found is less than the middle element, discard the latter half of the list, else discard the first half.
- Recursively look for the element until a singleton is reached. If this is the element, then return true/index of element, else return false/sensible error variable.

**Algorithm 2:** Binary Search

**Input**: Array $A$ of distinct increasing integers, and an integer $z$
**Output**: "Yes" if $z \in A$, "No" otherwise

```
1  def BIN-SEARCH (A, p, r, z)
2      if p ≥ r then
3          return "No"
4      else
5          q = ⌊(p + q)/2⌋;
6          if z = A[q] then
7              return "Yes"
8          else
9              if z < A[q] then
10                 BIN-SEARCH(A, p, q, z);
11             else
12                 BIN-SEARCH(A, q + 1, r, z);
```

**Running time** Let $T(n)$ be the worst-case time, therefore

$$T(n) = \begin{cases} O(1) & \text{if } n = 1 \\ T(\lfloor n/2 \rfloor) + O(1) & \text{otherwise} \end{cases}$$

By the Master Theorem, $T(n) = O(\log n)$. By looking at the "decision tree" of the binary search, it is clear that binary search is $\Omega(\log n)$, therefore, $T(n) \in \Theta(\log n)$.

### 2.1.3 Selection Problem

The $i^{\text{th}}$ ordered statistic of a set of $n$ distinct elements is the element that is larger than exactly $i-1$ other elements.

Upper-bound time: $O(n \log n)$ – sort the array in $O(n \log n)$ time and return the $i^{\text{th}}$ element.

It is possible to do so in worst case $O(n)$ time.

By partitioning the array, it is possible to recursively solve the above problem.

Steps 1,2, and 4 take $O(n)$ time.

The number of elements $\leq x$ is, at least $3\left(\lceil \frac{1}{2} \lfloor \frac{n}{5} \rfloor \rceil\right) \geq \lceil \frac{3n}{10} \rceil - 2$. Thus, in the worst case, `SELECT` is called on, at most $\lfloor 7n/10 \rfloor + 2$ elements, so

$$T(n) \leq T(\lfloor n/5 \rfloor) + T(\lfloor 7n/10 \rfloor + 2) + cn$$

for some constant $c \in \mathbb{R}$.

By supposing that $T(n) \in O(n)$, and then substituting $bn$ in for $T(n)$, for $b \geq 10cn/(n-20)$, $T(n) \leq cn$ else $T(n) \in O(1)$.

**Algorithm 3:** Partition algorithm

**Input**: An array $A$ of distinct numbers, with indices $p \leq q < r$ and $m = A[q]$ as the pivot
**Output**: An index $q'$ with $p \leq q' \leq r$ such that $A[p..r)$ is a permutation of $A$, $\forall a \in A[p..q') \Rightarrow a < m \wedge \forall a \in A[q'..r) \Rightarrow a \geq m \wedge A[q'] = m$.

**Algorithm 4:** Selection Algorithm

**Input**: An array $A$ of distinct numbers and the $i^{\text{th}}$ order statistic to find
**Output**: The $i^{\text{th}}$ smallest element

```
1  Divide the n input elements
   into groups of 5, with one group of n  mod 5 elements;
2  Find the median of the first ⌊n/5⌋ groups in O(1);
3  Find the median-of-medians x by calling select recursively;
4  Partition the input array
   around x, with the lower partition having k − 1 elements;
5  if i = k then return x;
6  else
7      if i < k then SELECT(A[0..k), i);
8      else SELECT(A[k + 1..n), i − k);
```

### 2.1.4 Integer Multiplication

By using an observation of Gauss', that is $(a+bi)(c+di) = ac - bd + (bc+ad)i$ can be done using just three multiplication operations ($ac$, $bd$, and $(a+b)(c+d)$), integer multiplication can then be performed much more efficiently. By splitting $n$-bit numbers $x$ and $y$ into left and right halves, each $n/2$ bits long, it is therefore possible to compute

$$\begin{aligned} xy &= \left(2^{n/2}x_L + x_R\right)\left(2^{n/2}y_L + y_R\right) \\ &= 2^n x_L y_L + 2^{n/2}\left(x_L y_R + x_R y_L\right) + x_R y_R \end{aligned}$$

Multiplication by $2^n$ can be implemented as a left-shift in constant time, and addition in linear time, giving the time $T(n)$ to multiply two $n$-bit numbers as

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

which, by the Master Theorem, gives $T(n) \in O(n^2)$.

By using Gauss' trick, and re-writing as $x_L y_L$, $x_R y_R$ and $(x_L + x_R)(y_L + y_R)$, the running time becomes

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

Again, by the Master Theorem, $T(n) \in \left(n^{\log_2 3}\right) \approx O\left(n^{1.59\ldots}\right)$.

### 2.1.5 Matrix Multiplication

Multiplying the $p \times q$ matrix $X$ by the $q \times r$ matrix $Y$ gives

$$Z_{ij} = \sum_{k=1}^{q} X_{ik} \cdot Y_{kj}$$

The above requires $p \times q \times r$ multiplications and $p \times (q-1) \times r$ additions. When $p = q = r = n$, there are $2n^3 - n^2 \in O(n^3)$ operations.

By dividing $X$ and $Y$ into quarters,

$$XY = \begin{bmatrix} A & B \\ C & D \end{bmatrix}\begin{bmatrix} E & F \\ G & H \end{bmatrix} = \begin{bmatrix} AE+BG & AF+BH \\ CE+DG & CF+DH \end{bmatrix}$$

It is then possible to recursively multiply the smaller matrices, and then add the individual elements in $O(n^2)$ time, giving a total running time of $T(n) = 8T(n/2) + O(n^2)$, which, by the Master Theorem, is $O(n^3)$.

By observing (similarly to Gauss) that

$$XY = \begin{bmatrix} P_5 + P_4 - P_2 + P_6 & P_1 + P_2 \\ P_3 + P_4 & P_1 + P_5 - P_3 - P_7 \end{bmatrix}$$

where

$$\begin{aligned} P_1 &= A(F - H) & P_5 &= (A + D)(E + H) \\ P_2 &= (A + B)H & P_6 &= (B - D(G + H) \\ P_3 &= (C + D)E & P_7 &= (A - C)(E + F) \\ P_4 &= D(G - E) \end{aligned}$$

Thus giving seven multiplications, hence $T(n) = 7T(n/2) + O(n^2) \Rightarrow T(n) \in O\left(n^{\log_2 7}\right) \approx O\left(n^{2.81}\right)$.

### 2.2 Dynamic Programming

By identifying and solving a collection of smaller subproblems, and initial problem can be solved by using the solution to the smaller problems in building up a complete solutions. Dynamic programming is not suited to solving all problems, as the solutions to the subproblems are not necessarily *compositional*, i.e. the optimal solution is not formed from a composite of the solutions of the subproblems.

Dynamic programming is only applicable when the *Principle of Optimality* is satisfied: "The optimal solution to a problem is a composite of optimal solutions to (some of) its subproblems".

#### 2.2.1 Knapsack Problems with Repetition

For the $n$ items, each of weight $w_1, w_2 \dots w_n$ and value $v_1, v_2, \dots v_n$ respectively, and a maximum carrying capacity of $W$ find the maximum possible value to carry without the total weight exceeding $W$. Define $K[w]$ to be the maximum achievable value with weight limit $w$. The solution to the problem is therefore

$$K[w] = \max\{K[w - w_i] + v_i : w_i \leq w\}$$

where $K[W]$ is the desired answer.

**Algorithm 5:** Knapsack Problem with repetition

**Input**: List of weights $\{w_1, w_2, \dots, w_n\}$ and respective values $\{v_1, v_2, \dots, v_n\}$, and maximum weight $W$
**Output**: Maximum value of items

```
1 K[0] = 0;
2 for w = 1 to W do K[w] = max{K[w − w_i] + v_i : w_i ≤ w};
3 return K[W];
```

The above fills an one-dimensional array of length $W + 1$, each entry taking $O(n)$ time, hence the total running time of the algorithm is $O(nW)$.

#### 2.2.2 Knapsack Problems without Repetition

For the $n$ items, each of weight $w_1, w_2 \dots w_n$ and value $v_1, v_2, \dots v_n$ respectively, and a maximum carrying capacity of $W$ find the maximum possible value to carry without the total weight exceeding $W$, and with all items being distinct. Define $K[w,]$ to be the maximum achievable value with weight limit $w$, choosing from items $1, 2 \dots j$, as $j$ varies between $0 \leq j \leq n$. The solution to the problem is therefore

$$K[w, j] = \max\{K[w - w_j, j - 1] + v_j, K[w, j - 1]\}$$

**Algorithm 6:** Knapsack Problem without repetition

**Input**: List of weights $\{w_1, w_2, \dots, w_n\}$ and respective values $\{v_1, v_2, \dots, v_n\}$, and maximum weight $W$
**Output**: Maximum value of items

```
1 for j = 0 to n do K[0, j] = 0;
2 for w = 0 to W do K[w, 0] = 0;
3 for j = 1 to n do
4     for w = 1 to W do
5         if w_j > w then
6             K[w, j] = K[w, j − 1];
7         else
8             K[w] = max{K[w − w_i] + v_i : w_i ≤ w}
9 return K[W, n];
```

where $K[W, n]$ is the desired answer.

The above fills an one-dimensional array of length $W + 1$, each entry taking $O(n)$ time, hence the total running time of the algorithm is $O(nW)$.

```
knapsack_spec
     :: [(Integer,Integer)] → Integer → Integer
knapsack_spec wvs w =
      maximum ( map (sum ∘ map snd) (
      filter ((≤ w) ∘ sum ∘ map fst) (
      subsequences wvs )))

subsequences :: [a] → [[a]]
subsequences = foldr f [[]]
  where f x = foldr (λy zs → (x:y):y:zs) []

knapsack_rec [] w = 0
knapsack_rec ((wi,vi):wvs) w
  | wi > w = knapsack_rec wvs w
  | otherwise = max (knapsack_rec
       wvs w) (knapsack_rec wvs (w-wi) + vi)

knapsack_dp wvs wtot = table ! (wtot,n)
  where n = length wvs
        table = array ((0,0),(wtot,n)) [((w,
            j), ks w j) | w ← [0..wtot], j ← [0..n]]
        ks w 0 = 0
        kw w j = if wj > w then table ! (w,j-1)
                 else max (table ! (w,j-1))
                   ((table ! (w-wj, j-1)) + vj)
          where (wj,vj) = wvs !! (j-1)
```

#### 2.2.3 Change Making Problem

Assuming an unlimited supply of coins, what is the minimum number of coins needed to give change to value $v$ using denominations $1 = x_1, x_2, \dots, x_n$?

By setting $C[u]$ to be the minimum number of coins required to give change to a total value of $u$, and looking for $C[v]$, the following recurrence is constructed:

$$\begin{aligned} C[0] &= 0 \\ C[u] &= 1 + \min\{C[u - x_1] : 1 \leq i \leq n \wedge u \geq x_i\} \end{aligned}$$

The above fills an one-dimensional array of length $v$ with each entry taking, at most $O(n)$ time, hence the total running time is $O(nv)$.

#### 2.2.4 Edit Distance Problem (Levenshtein Distance)

The edit distance of a string is the "minimum number of edits needed to transform one string into another" where an edit is an insertion, deletion or substitution.

**Algorithm 7:** Change giving algorithm

**Input**: List of coin denominations
$1 = x_1, x_2, \ldots, x_n$, and value of change $v$
**Output**: Minimum number of coins required to give change

```
C[0] = 0;
for u = 1 to v do
    C[u] = 1 + min{C[u − x_1] : 1 ≤ i ≤ n ∧ u ≥ x_i};
return C[v];
```

For the strings $x[0..m)$ and $y[0..n)$, let $0 \leq i \leq m$ and $0 \leq j \leq n$, set $E[i,j]$ to be the edit distance between $x[0..i)$ and $y[0..j)$, and find $E[m,n]$.

There are three cases to be considered:

1. Cost= 1, to align $x[0..i-1)$ with $y[0..j)$ (insertion)
2. Cost= 1, to align $x[0..i)$ with $y[0..j-1)$ (deletion)
3. Cost= 1 if $x[i] \neq y[j]$ and 0 otherwise, to align $x[0..i-1)$ with $y[0..j-1)$.

By letting $\delta(i,j) := 1$ if $x[i] \neq y[j]$ and 0 otherwise, then

$$E[i,j] = \min\{E[i-1,j]+1, E[i,j-1]+1, E[i-1,j-1]+\delta(i,j)\}$$

**Algorithm 8:** Levenshtein Distance

**Input**: Strings $x[0..m)$ and $y[0..n)$
**Output**: Edit distance between $x$ and $y$

```
for i = 0 to m do E[0, i] = i;
for j = 0 to n do E[j, 0] = j;
for i = 1 to m do
    for j = 1 to n do
        E[i, j] = min{E[i −
        1, j] + 1, E[i, j − 1] + 1, E[i − 1, j − 1] + δ(i, j)}
return E[m, n];
```

### 2.2.5 Travelling Salesman Problem

For the complete undirected graph with vertex-set $\{0,1,\ldots,n-1\}$ and edge lengths stored in the matrix $D = (d_{ij})$. Find a tour starting and ending at a specified node with minimum total length including all other vertices exactly once. This problem is *NP-hard*, as it is unlikely to ever be solved in polynomial time. A brute force technique of examining every path takes $O(n!)$ $((n-1)!$ possibilities), but dynamic programming reduces this to $O(n^2 2^n)$.

By considering the subset $\{0,1,\ldots,j\} \subseteq S \subseteq \{0,1,\ldots,n-1\}$, let $C[S,j]$ but the shortest simple path length starting at 0 and ending at $j$, visiting each node in $S$ exactly once. For $|S| > 1$, set $C[S,0] = \infty$ (simple graph, therefore cannot start and end at same node). By expressing $S$ in terms of its subproblems:

$$C[S,j] = \min\{C[S \setminus \{j\}, i] + d_{ij} \mid i \in S \wedge i \neq j\}$$

The required answer is therefore

$$\min\{C[\{O,1,\ldots,n-1\},j] + d_{j0} \mid 0 \leq j < n\}$$

There are, at most $n \cdot 2^n$ subproblems, each taking linear time to solve, giving a total running time of $O\left(n^2 2^n\right)$.

### 2.2.6 All-pairs shortest path

Given a directed graph $(V,E)$ with weight (considered as distance) $w : E \mapsto \mathbb{R}^{\geq 0}$, for each pair of vertices $u$ and $v$, find the shortest path from $u$ to $v$.

Suppose the vertex-set is $\{0,1,\ldots,n-1\}$ and let $d[i,j;k] =$ length of shortest path from $i$ to $j$, all of whose intermediate nodes are taken from $[0..k)$.

Initially,

$$d[i,j;0] = \begin{cases} w(i,j) & \text{if } (i,j) \in E \\ \infty & \text{otherwise} \end{cases}$$

If we have $d[i,k;k]$ and $d[k,j;k]$ then the shortest path that from $i$ to $j$ that uses $k$, as well as other nodes, goes through $k$ once, assuming no negative cycles. Hence, $k$ is used in a shortest path from $i$ to $j$ iff $d[i,k;k] + d[k,j;k] < d[i,j;k]$, hence $d[i,j;k+1]$ should be updated accordingly. The running time is $O\left(|V|^3\right)$.

**Algorithm 9:** Floyd-Warshall Algorithm

**Input**: The directed graph $(V,E)$
with weight (considered as distance) $w : E \mapsto \mathbb{R}^{\geq 0}$
**Output**: Shortest path between all pairs of nodes

```
for i = 0 to |V| − 1 do
    for j = 0 to |V| − 1 do d[i, j; 0] = ∞;
forall the edge (i, j) ∈ E do d[i, j; 0] = w(i, j);
for k = 0 to |V| − 1 do
    for i = 0 to |V| − 1 do
        for j = 0 to |V| − 1 do
            d[i, j; k+1] = min{d[i, k; k]+d[k, j; k], d[i, j; k]};
return d;
```

## 2.3 Greedy Algorithms

Similar to dynamic programming algorithms, these are also used to solve optimisation problems, and work by only choosing the step with the most immediate benefit as the next step, without looking ahead, or reconsidering earlier decisions. They have the advantage that they are often more simple to implement, and there is no need for large amounts of storage, as only one decision is taken at each stage, and that decision is never reconsidered.

### 2.3.1 Change Making Algorithms

**The Greedy Approach** Start with no change, and at each stage, choose a coin of the largest denomination available that does not exceed the balance to be paid.

However, the above method does not work with all denominations of coins, and does not always yield the optimal solution. For example, with $100, 60, 50, 5, 1$, to pay 110, the greedy algorithm would give 3, $(100 + 5 + 5)$, whereas $60 + 50$ is a more optimal solution.

### 2.3.2 Minimum Spanning Tree Algorithm

**Algorithm 10:** Generic MST algorithm

```
A = ∅;
while A is not a spanning tree do
    find an edge (u, v) that is safe for A;
    A = A ∪ {(u, v)};
return A
```

**Loop Invariant** : "$A$ is safe, i.e. a subset of some MST"
**Initialisation** : The invariant is trivially satisfied by $A = \emptyset$
**Maintenance** : Since only safe edges are added, $A$ remains a subset of some MST.

**Termination** : All edges added to $A$ are in an MST so $A$ must be a spanning tree that is also minimal.

If $(S, V \setminus S)$ is a cut that respects $A$, and $(u, v)$ is a light edge crossing the cut, then $(u, v)$ is safe for $A$.

#### 2.3.3 Kruskal's Algorithm

**Description**

- Start with each vertex being its own connected component.
- Find the edge with the lowest weight.
- Merge two components by choosing the light edge connecting them

Kruskal's requires a disjoint-set data structure to be most effective. It is a set of disjoint sets $\mathcal{S} = \{S_1, \ldots, S_k\}$ where each set is represented by an individual element in each set.

**Algorithm 11:** Kruskal's Algorithm

```
   Input: The graph (V, E) and weight function w : E ↦ ℝ
   Output: An MST for the graph
1  A = ∅;
2  forall the v ∈ V do MAKE-SET(v);
3  Sort E by increasing weight w;
4  forall the edges (u, v) from the sorted list do
5      if FIND-SET(u)≠FIND-SET(v) then
6          A = A ∪ {(u, v)};
7          UNION(u, v);
8  return A;
```

**Invariant** Let $\mathcal{S}$ be the collection of sets in the disjoint-set data structure and $L$ be the sorted list of edges not yet processed by the for-loop.

1. $A$ is safe
2. For each $C \in \mathcal{S}$, $(C, A \restriction C)$ is a spanning tree of the subgraph $(C, E \restriction C)$.
3. Every processed edge's start- and end-points are in the same set in $\mathcal{S}$.

**Initialisation** $A = \emptyset$, $\mathcal{S}$ consists of only singleton sets and no edge has been processed $E \setminus L = \emptyset$ (hence all trivially true).

**Maintenance** Let $e = (u, v)$ be the edge to be processed, $C_1$ and $C_2$ be the sets that $u$ and $v$ belong to respectively, and $A$, $\mathcal{S}$ and $L$ refer to the state before the iteration, and $A'$, $\mathcal{S}'$ and $L'$ be the state after the iteration.

If $e$ is included in $A$, then $C_1$ and $C_2$ are different, and $e$ is the minimum edge crossing the cut $(C_1, V \setminus C_1)$, and the cut respects $A$. As $e$ is the next element to be processed, it must also be the lightest element, hence the cut lemma holds. The union operation ensures that the third part of the invariant holds.

If $e$ is not to be included in $A$, then there are no changes to $A$, $\mathcal{S}$ and $L$ is updated to include the discarded edge.

**Termination** All edges have been processed, therefore, $L = \emptyset$, and since all the nodes belong to the same set $C \in \mathcal{S}$, $C$ spans the whole graph, and by application of the cut lemma, and safe edges is an MST.

**Running Time** Initialisation of $A$ takes $O(1)$, first for-loop calls `MAKE-SET` $|V|$ times. $E$ is sorted in $|E| \log |E|$ time. The second for-loop has $2|E|$ calls to `FIND-SET` and $|V| - 1$ calls to `UNION`, giving a total running time of $O(|E| \log |E|)$.

#### 2.3.4 Prim's Algorithm

**Description** By growing the MST $A$ from a given root node $r$, at each stage, find a light edge crossing the cut $(V_A, V \setminus V_A)$ where $V_A$ is the edges incident on $A$.

The lightest edge can be found quickly by using a priority queue, where each entry in the queue is a vertex in $V \setminus V_A$. $key[v]$ is the minimum weight of any edge $(u, v)$ where $v \in V_A$, the vertex returned by `EXTRACT-MIN` is $v$ such that $\exists u \in V_A$ where $(u, v)$ is a light edge crossing $(V_A, V \setminus V_A)$. $key[v] = \infty$ if $v$ is not adjacent to any vertex in $V_A$.

**Algorithm 12:** Prim's Algorithm

```
   Input: The graph (V, E) and weight function w : E ↦ ℝ
   Output: An MST for the graph
1  Q = ∅;
2  forall the u ∈ V do
3      key[u] = ∞, π[u] = †;
4      INSERT(Q, u);
5  DECREASE-KEY(Q, r, 0) while Q ≠ ∅ do
6      u = EXTRACT-MIN(Q);
7      forall the v ∈ Adj[u] do
8          if v ∈ Q ∧ w(u, v) < key[v] then
9              π[v] = u, DECREASE-KEY(Q, v, w(u, v));
```

**Running Time** Initialising $Q$ takes $O(1)$, the first loop runs in $O(|V|)$, changing priority of $r$ takes $O(\log |V|)$, and $|V|$ `EXTRACT-MIN` calls are required with, at most $|E|$ `DECREASE-KEY` operations, giving running time of $O(|E| \log |V|)$. The graph is connected, so $O(\log |E|) = O(\log |V|)$, hence total running time of $O(|E| \log |V|)$.

### 2.4 Dynamic Programming vs. Divide-and-Conquer

Dynamic programming is an optimisation technique, whereas divide-and-conquer is not normally used to solve optimality problems.

Both techniques split the input problems into smaller parts and use the solutions to the smaller parts to form a larger solution, however, dynamic programming solves the subproblems at all split points, whereas divide-and-conquer uses pre-determined split points using non-overlapping problems. Dynamic programming uses solutions to already calculated subproblems to find the total solution, to reduce space complexity.

## 3 Data Structures

### 3.1 Heaps

A heap is a type of tree without explicit pointers. Each level is filled from left to right, and the next level is only added when the previous is full. All heaps satisfy either the max-heap or min-heap property: "the value of a node (except the root node) is less than (greater than) or equal to that of its parent". In general, a heap can have any number of children on each of its nodes, and the maximum/minimum element of a max-/min-heap is at the root. Heaps are used as efficient priority queues, and for heapsort, which has a complexity of $O(n \log n)$.

#### 3.1.1 Representation

The root is always at $A[0]$, and for any node $i > 0$, its parent is at $A[\lfloor (i-1)/2 \rfloor]$ and its left and right children are at $A[2i+1]$ and $A[2i+2]$.

#### 3.1.2 Maintaining heaps

**Running Time** $\Theta(1)$ to find the largest of node and children. Worst-case has tree with last row half full (i.e. subtree rooted at $i$ has, at most $2n/3$ elements), so $T(n) = T(2n/3) + \Theta(1) \Rightarrow T(n) = O(n^0 \log_{3/2} n) = O(\log n)$ by the Master Theorem.

**Algorithm 13:** Heapify algorithm

**Input**: Tree with left and right sub-trees of $i$ stored as heaps
**Output**: $A$ where entire tree is also a heap.

```
1 n = A.heapsize;
2 l = 2i + 1, r = 2i + 2;
3 if l < n ∧ A[l] > A[i] then largest = l else largest = i;
4 if r > n ∧ A[r] > A[largest] then largest = r;
5 if largest ≠ i then
6     exchange A[i] with A[largest];
7     HEAPIFY(A, largest);
```

**Algorithm 14:** Make-Heap algorithm

**Input**: An unsorted integer array $A$ of size $n$
**Output**: A heap of size $n$

```
1 A.heapsize = A.length;
2 for i = ⌊A.length/2⌋ to 0 do HEAPIFY(A, i);
3 return A
```

**Correctness**

**Invariant** : each node $i+1, i+2, \ldots, n-1$ is the root of a heap for $-1 \leq i \leq \lfloor n/2 \rfloor$

**Initialisation** : each node $\lfloor n/2 \rfloor, \lfloor n/2 \rfloor + 1, \ldots, n-1$ is a leaf, which is the root of a trivial heap, therefore the invariant holds.

**Maintenance** : calling HEAPIFY$(A, i)$ causes $i$ to become the root of a new heap, hence, when $i$ is decremented, nodes at $i+1, i+2, \ldots, n-1$ are all roots of heaps.

**Termination** : when $i = -1$, the element at 0 is the root of a heap, therefore all elements below it are also roots of heaps.

**Running Time** There are $n$ calls to HEAPIFY, each taking $O(\log n)$ time, giving $O(n \log n)$.

As HEAPIFY is linear with the height of the node that it runs on, the height of the heap is $\lfloor \lg n \rfloor$, hence the cost of MAKE-HEAP is

$$\sum_{h=0}^{\lfloor \lg n \rfloor} \left\lceil \frac{n}{2^{h+1}} \right\rceil O(h) = O\left( n \sum_{h=0}^{\lfloor \lg n \rfloor} \frac{h}{2^h} \right)$$

As $\sum_{i=0}^{\infty} \frac{i}{2^i} = 2$, the total running time is therefore $O(n)$.

### 3.1.3 Heap-Sort

**Description** From a given input array, build a max-heap, and starting from the root element, swap the root node (maximum node) with the node at the end of the heap, and call HEAPIFY to maintain the heap property. Repeat until the heap's size is 1, when the root node is the minimal element, and the array is therefore sorted.

**Algorithm 15:** Heap-Sort

**Input**: The unsorted array $A$ with distinct elements
**Output**: Sorted permutation of $A$

```
1 MAKE-HEAP(A) for i = A.heapsize − 1 to 1 do
2     swap A[0] with A[i];
3     A.heapsize = A.heapsize − 1;
4     HEAPIFY(A, 0);
5 return A
```

**Running Time** MAKE-HEAP runs in $O(n)$, the for-loop is run $n-1$ times, with the swap operation, and decrementing the heap size taking $O(1)$. HEAPIFY runs in $O(\log n)$ time, thus meaning the entire algorithm runs in $O(n \log n)$.

**Advantages** Heap-sort is a stable sort that runs in $O(n \log n)$ time. In practice, however, a well implemented quicksort can beat heap-sort.

## 3.2 Priority Queues

A priority queue is an abstract data structure, to maintain a sequence a values, each with an associated key or weight.

Priority queues normally have the following associated operations:

- INSERT$(S, x, k)$ to insert the element $x$ with key $k$ into the queue $S$.
- MAXIMUM$(S)$ to find the element in $S$ with the largest key. Minimum priority queues have an corresponding function MINIMUM.
- EXTRACT-MAX$(S)$ to find and remove the element in $S$ with the largest key. Minimum priority queues have an corresponding function EXTRACT-MIN.
- INCREASE-KEY$(S, x, k)$ to increase the value of $x$'s key to $k$. The new value of $k$ must be larger than the old value. Minimum priority queues have an corresponding function DECREASE-KEY.

Priority queues are often found used to schedule jobs, in Djikstra's shortest path algorithm and Prim's MST algorithm.

### 3.2.1 Implementation

It is possible to implement priority queues using an array or a doubly-linked list, however, the implementation of the extraction functions run in $\Theta(n)$ time, which can be improved with a heap implementation.

Implementing a priority queue using a heap allows the extraction operation to run in $O(\log n)$ time, however, it increases the time to insert an element from $O(1)$ to $O(\log n)$, but this is often seen as an acceptable trade-off.

The implementations provided all assume a maximum priority queue. The running time is $O(\log n)$, as all calls, except the call to HEAPIFY run in $O(1)$ time. Running time: $O(\log n)$

**Algorithm 16:** Return the maximal element of the queue

**Input**: The priority queue $A$
**Output**: The maximum value of $A$

```
1 return A[0];
```

**Algorithm 17:** Extract Max Algorithm

**Input**: The priority queue $A$
**Output**: The old maximum value $A$

```
1 if A.heapsize < 1 then error "heap underflow";
2 max = A[0];
3 A.heapsize = A.heapsize − 1;
4 A[0] = A[A.heapsize];
5 HEAPIFY(A, 0);
6 return max;
```

## 3.3 Queues

A FIFO queue is an abstract data structure, with the following methods:

ENQUEUE$(Q, x)$ – inserts $x$ at the end of the queue

**Algorithm 18:** Increase Key Algorithm

**Input**: The priority queue $A$, the value to increase $i$, and the new key $k$
**Output**: The new priority queue $A$

```
1 if k < A[i]
  then error "new key is smaller than current key";
2 A[i] = k;
3 while i > 0 ∧ A[i.parent] > A[i] do
4     swap A[i] and A[i.parent];
5     i = i.parent;
6 return A
```

**Algorithm 19:** Insertion Algorithm

**Input**: The priority queue $A$ and the key $k$
**Output**: The new priority queue $A$

```
1 A.heapsize = A.heapsize +1;
2 A[A.heapsize −1] = −∞;
3 HEAP-INCREASE-KEY(A, A.heapsize −1, k);
4 return A;
```

`DEQUEUE`$(Q)$ – finds and removes the element at the head of the queue $Q$.

`ISEMPTY`$(Q)$ – tests if the queue $Q$ is empty. Often abbreviated to $Q = \emptyset$ in pseudo-code.

### 3.3.1 Implementation

Linked list with an extra pointer to the tail of the list, with the head of the queue being the head of the linked list, making all operations $O(1)$.

## 3.4 Graphs

Graphs are often used to form abstractions to certain problems, for example, colouring a map to ensure two adjacent countries do not have the same colour is easier when the map is considered as a graph, where each country is a node, and two nodes are linked iff they share a border. Graphs can also be used to help with exams scheduling: each exam is represented by a node, and joined if they are both taken by at least one student.

### 3.4.1 Definitions

**Directed graph** $(V, E)$, where $V$ is a set of nodes and $E \subseteq V \times V$ of edges. If $u$ is connected to $v$ (does not imply $v$ is connected to $u$) by the edge $e = (u, v)$, we say that $e$ is incident on $u$ and $v$.

**Path** from a vertex $u$ to $u'$ is a sequence $\langle v_0, v_1, \ldots, v_k \rangle$, with length $k$, of vertices such that $u = v_0$, $u' = v_k$ and each $(v_i, v_{i+1}) \in E$.

**Cycle** is a path such that $v_0 = v_k$ and the path contains at least one edge. Self-loops are cycles with length 1. Simple cycles are cycles with all nodes being distinct.

**Acyclic** graph are directed graphs containing no cycles.

**Strongly connected** graphs are graphs that, for any two nodes, there is a path from the first to the second.

**Undirected** graphs are graphs where $E$ is symmetric ($(u, v) \in E \Leftrightarrow (v, u) \in E$)

### 3.4.2 Representations

Graphs can be represented as an adjacency matrix (where the entry $a_{ij} = 1$ if $(v_i, v_j) \in E$ and 0 otherwise. An edge can be found in constant time, and requires a storage size of $O(|V|^2)$. The undirected graph's representation is symmetric.

Adjacency lists consist of $|V|$ linked lists, with the list for vertex $u$ containing links to all vertices that $v$ is adjacent to. The data structure cannot be checked in constant time, and has size $O(|V| + |E|)$. Undirected graphs are represented by lists where $u$ is in $v$'s list iff $v$ is in $u$'s.

### 3.4.3 Depth-first Search (DFS)

Linear time algorithm to explore a graph by exploring all the reachable unseen vertices from each vertex.

As DFS progresses, each nodes is assigned a changing colour.

**Black** finished – all reachable vertices have been discovered

**Grey** discovered but not finished

**White** not yet discovered

**Algorithm 20:** Depth-First Search Algorithm

**Input**: Graph $G = (V, E)$
**Output**: Discovery and finishing time for each vertex $v \in V$

```
1  def DFS(V, E)
2      forall the u ∈ V do colour[u]=WHITE;
3      t = 0;
4      forall the u ∈ V do
5          if colour[u]=WHITE then
6              DFS-VISIT(u);
7  def DFS-VISIT(u)
8      colour[u]=GREY;
9      t = t + 1;
10     d[u]= t;
11     forall the v ∈ Adj[u] do
12         if colour[u]=WHITE then
13             DFS-VISIT(u);
14     colour[u]=BLACK;
15     f[u]= t;
```

**Running Time** The first loop runs in $\Theta(|V|)$. `DFS-VISIT` is called once for each $v \in V$, since it is not called on grey or black vertices. During `DFS-VISIT`, the loop is run $|\,Adj[v]|$ times. As $\sum_{v \in V} |\,Adj[v]| = \Theta(|E|)$, the total running cost of `DFS-VISIT`, as $v$ changes over $V$ is $\Theta(|E|)$, thus giving a total running time of $O(|V| + |E|)$ for depth-first search.

**Depth-First Search Forest** By analysing the arrays produced by DFS, a number of DFS trees can be produced, making up a DFS forest. Each tree is made up of edges $(u, v)$ such that $u$ is grey and $v$ is white when $(u, v)$ is first explored. If a node $u$ is a descendent of $v$ in the DFS forest, it is also a descendent of $v$ in the original graph.

By using the following shorthand:

$$\begin{array}{cccc} d[u] & f[u] & d[v] & f[v] \\ (_u & )_u & (_v & )_v \end{array}$$

and the convention the value associated with the preceding bracket is less than the next. The following rules are derived:

$(_u\ )_u\ (_v\ )_v$ Neither $u$ nor $v$ are descendants of each other.

$(_u\ (_v\ )_v\ )_u$ $v$ is a descendent of $u$.

$(_u\ (_v\ )_u\ )_v$ cannot happen.

There are four different types of edge in a DFS forest:

**Tree** – edges of the DFS forest

**Back** – lead from a node to an ancestor

**Forward** – lead from a node to a non-child descendent

**Cross** – can link from either same or different trees. Always lead to edges with earlier discovery times, and only exist in directed graphs.

A directed graph has a cycle iff its DFS forest contains a back edge.

### 3.4.4 Topological Sort

The topological sort of a directed acyclic graph is a total order of the vertices such that if $(u, v) \in E$ then $u > v$. The running time of the above is $\Theta(|V| + |E|)$.

**Algorithm 21:** Topological Sort

**Input**: A directed acyclic graph $G = (V, E)$
**Output**: Elements of $V$ in topological order
1 Call `DFS`$(V, E)$ to find $\mathsf{f}[v]$ for $v \in V$;
2 Output vertices in order of decreasing finishing times;

### 3.4.5 Strongly Connected Components

In directed graphs, two vertices $u$ and $v$ are strongly connected if there is a path, both from $u$ to $v$ and from $v$ to $u$. A strongly connected component is the largest possible set of vertices that are all connected.

For the directed graph $G = (V, E)$, the transpose is $G^T = (V, E^T)$, where $E^T = \{(v, u) \mid (u, v) \in E\}$. $G^T$ can be computed in $\Theta(|E| + |V|)$ using adjacency lists. $G$ and $G^T$ both have the same strongly connected components, which can be used to identify the SCC of $G$.

**Algorithm 22:** Strongly Connected Component Discovery

**Input**: The directed graph $G$
**Output**: Elements of each SCC in $G$ outputted in turn
1 Call `DFS`$(G)$ to find $\mathsf{f}[u]$ for all $u$;
2 Compute $G^T$;
3 Call `DFS`$(G^T)$, but in the loop of `DFS`, order by decreasing $\mathsf{f}[u]$ as computed above;
4 Output the vertices in each tree of the DFS forest as found above;

### 3.4.6 Breadth-First Search (BFS)

Linear time algorithm to explore a graph by exploring all the reachable unseen vertices from each vertex.

By sending out a 'wave' from a source edge $s$, all nodes 1 edge from $s$ will be discovered, then all nodes 2 edges away, etc. A queue is used to maintain the wavefront, with the property that $v \in Q \Leftrightarrow$ wave has hit $v$, but has not yet moved past it. The initial setup records that $s$ can be reached from itself in 0 time and in 0 steps, and that, as far as we know, all other vertices cannot be reached, and therefore take an infinite amount of time to be reached. In the `for`-loop, the next unfinished node is examined. All adjacent nodes that have not yet been seen have their shortest distance recorded and their backpointer set.

**Algorithm 23:** Bread-First Search

**Input**: The graph $G = (V, E)$ and a source node $s \in V$
**Output**: The discovery time and shortest distance from $s$ for each node $v$
1 $\mathsf{d}[v] = 0$, $\pi[v] = \mathsf{null}$;
2 **forall the** $u \in V \setminus \{s\}$ **do** $\mathsf{d}[u] = \infty$, $\pi[u] = \mathsf{null}$;
3 $Q = \emptyset$;
4 `ENQUEUE`$(Q, s)$;
5 **while** $Q \neq \emptyset$ **do**
6 $\quad u =$ `DEQUEUE`$(Q)$;
7 $\quad$ **forall the** $v \in Adj[u]$ **do**
8 $\quad\quad$ **if** $\mathsf{d}[v] = \infty$ **then**
9 $\quad\quad\quad \mathsf{d}[v] = \mathsf{d}[u] + 1$;
10 $\quad\quad\quad \pi[v] = u$;
11 $\quad\quad\quad$ `ENQUEUE`$(Q, v)$;

**Running Time** $O(|V| + |E|)$. $O(|V|)$ as every vertex is enqueued at most once, $O(|E|)$ as every vertex dequeued at most once, and $(u, v)$ is examined only when $u$ is dequeued, therefore every edge is examined at most once if the graph is directed, and twice is the graph is undirected.

Unlike DFS, BFS may not reach all the vertices, as it only works outwards from a source vertex. If there is no path from $s$ to $v$, then it will not be discovered.

### 3.4.7 Shortest Path Problems

Consider a directed graph $G = (V, E)$ with weight or length $w : E \mapsto \mathbb{R}$.

The weight of a path $p = \langle v_0, v_1, \ldots, v_k \rangle$ is given as

$$w(p) := \sum_{i=1}^{k} w(v_{i-1}, v_i)$$

The shortest-path weight from $u$ to $v$ is given as

$$\delta(u, v) := \begin{cases} \min\{w(p) : p \text{ is a path from } u \text{ to } v\} & \exists \text{ path from } u \text{ to } v \\ \infty & \text{otherwise} \end{cases}$$

A shortest path from $u$ to $v$ is a path such that $\delta(u, v) = w(p)$. There are a number of different shortest path algorithms:

**Dynamic Programming** – solves the single-source problem, for a DAG in $\Theta(|E| + |V|)$

**BFS** – solves the single-source problem for graphs with unit weight ($\forall e \in E \cdot w(e) = 1$) in $O(|V| + |E|)$.

**Djiksktra** – solves the single-source problem for directed or undirected graphs with non-negative weights in $O\left((|V| + |E|) \log |V|\right)$

**Floyd-Warshall** – solvers the all-pairs shortest path probelm for graphs without negative weight cycles in $O\left(|V|^3\right)$

### 3.4.8 Djikstra's Algorithm

Using a priority queue, Djikstra's adapts BFS to construct the shortest-path tree from $\pi[v]$.

By maintaining two sets of vertices – $S$ for vertices where the shortest-path is known, and $Q = V \setminus S$ to be processed (stored in the priority queue). To check for the presence of a unique shortest path, create an aray $u$, initialised to `true`. By modifying the `if`-clause in the inner `for`-loop, if there is a new shortest path for a node $v$, then the existence of a new shortest path is dependent on the existence of a shortest path to its parent. If a shortest path has already been discovered, and another is found, then $u[v] =$ `false`.

**Algorithm 24:** Djikstra's Shortest Path Algorithm

**Input**: The graph $G = (V, E)$,
$s \in V$ as the source node, $w : E \mapsto \mathbb{R}_{\geq 0}$
**Output**: For each $v \in V$, $d[v]$ and $\pi[v]$

```
1  forall the v ∈ V do d[v] = ∞, π[v] = null;
2  d[s] = 0; S = ∅;
3  Q = MAKE-QUEUE(V) with d[v] as keys;
4  while Q ≠ ∅ do
5      u = EXTRACT-MIN(Q);
6      S = S ∪ {u};
7      forall the v ∈ Adj[u] do
8          if d[u] + w(u, v) < d[v] then
9              d[v] = d[u] + w(u, v);
10             π[v] = u;
11             DECREASE-KEY(Q, v, d[v]);
```

### 3.4.9 Floyd-Warshall

See section 2.2.6

### 3.4.10 Bellman-Ford

This is used to test for the presence of a negative weight cycle from a specific node.

**Algorithm 25:** Bellman-Ford Algorithm

**Input**: The graph
$G = (V, E)$, $s \in V$ as a source node, $w : E \mapsto \mathbb{R}$
**Output**: False if a negative weight cycle
can be found from $s$, else $d[v]$, $\pi[v]$ for all $v \in V$

```
1  forall the v ∈ V do d[v] = ∞; π[v] = null;
2  d[s] = 0;
3  for i = 1 to |V| − 1 do
4      forall the (u, v) ∈ E do
5          if d[u] + w(u, v) < d[v]
           then d[v] = d[u] + w(u, v); π[v] = u;
6  forall the (u, v) ∈ E do
7      if d[u] + w(u, v) < d[v] then return TRUE ;
8  return FALSE;
```

**Running Time** The initialisation iterates over $V$, so takes $\Theta(|V|)$. Each of the iterations of the second loop then iterate over the edges in $E$, giving a runtime of $O(|V|\Theta(|E|))$, and the final loop requires $O(|E|)$. Therefore, the total runtime is $O(|V||E|)$.