Data Structures & Algorithms - Notes

Dom Hutchinson

December 18, 2019

Contents

1 Theory

1.1 Strong Induction

Definition 1.1 - *Strong Induction*
Strong Induction is a technique used for proving a statement $P(n),\ n \in \mathbb{N}$ is true $\forall\ m \in \mathbb{N}^{>n_0}$, for some $n_0 \in \mathbb{N}$.

Proposition 1.1 - *Strong Induction Method*
Let $P(n)$ be a statement and $b_0, \dots, b_i$ be the base cases.
To prove the statement $P(n)$ is true for $n = b_i, \dots, \infty$ do the following

1. Prove $P(b_0), \dots, P(b_i)$ are true.
2. For the sake of contradiction, let $\exists\ m \in \mathbb{N}$ be the smallest integer such that $P(m)$ is false. Then $\forall\ n \in \mathbb{N}$ st $b_i \leq n < m$, $P(n)$ is true.
3. Show that having all of these cases of n being true means that $P(m)$ must be true. Then we have a contradiction and $P(n)$ is true $\forall\ n \in \mathbb{N}_{\geq b_i}$.

Example 1.1 - *Strong Induction*
Let $P(n)$ be the statement that postage of n cents can be formed using just 4-cent & 5-cent stamps.
Prove $P(n)$ is true $\forall\ n \in \mathbb{N}^{\geq 12}$.
Base Cases.

$$\begin{array}{rclrcl} P(12) & = & 4+4+4 & P(13) & = & 4+4+5 \\ P(14) & = & 4+5+5 & P(15) & = & 5+5+5 \end{array}$$

$P(n)$ holds for the base cases of $n \in \{12, 13, 14, 15\}$.
Inductive Hypothesis.
$P(j)$ is true $\forall\ j \in \mathbb{N}$ where $12 \leq j \leq k$ for $k \geq 15$.
Inductive Step.
By the inductive hypothesis we know $P(k-3)$ holds since $k - 3 \geq 12$.
To form postage for $k+1$ cents, add a 4-cent to the postage for $k-3$ cents.
Hence, by the theorem of strong induction $P(n)$ holds $\forall n \in \mathbb{N}^{\geq 12}$.

Remark 1.1 - *Usefulness of Strong Induction*
Strong induction is useful as it can be used to prove algorithms work upto an infinite amount of data.
This means we don't need to work about the effects of scaling on the algorithm's result.

1.2 Invariant

Definition 2.1 - *Invariants*
An *Invariant* is a function or quantity which remains unchanged after a transformation has been applied.

Definition 2.2 - *Preserved Invariant*
A *Preserved Invariant* is a statement, P, about the state of a finite state machine such that if $P(q)$ is true then $P(r)$ is true $\forall$ states r which occur after q.
N.B. - These are sometimes referred to as a *loop invariant.*

Proposition 2.1 - *Preserved Invariant Proof Method*
To prove the invariance of a loop do the following

i) Find a suitable invariant P.

ii) *Initialisation* - Prove P is true for the initial state.

iii) *Maintenance* - Prove that if a state is true then the subsequent state must be true.

iv) *Termination* - Prove the loop eventually terminates & produces a useful property when it does.

Example 2.1 - *Invariant*
Consider the following algorithm

```
SUM(A)
a=0
for (i=0; i<A.length; i++):
  a+=A[i]
return a
```

Here we shall prove that the statement *"At the start of iteration j of the loop, the variable a contains the sum of the numbers in the subarray $A[0:j]$"* is a loop invariant.
Initialisation
At the start of the first loop $a = 0$.
The subarray $A[0:0]$ contains no elements so sums to 0.
Thus the invariant holds.
Maintenance
Assume the loop invariant holds at the start of the j^{th} iteration.
Then it must be that a contains the sum of the subarray $A[0:j]$.
In the body of the loop we add $A[j]$ to a.
Thus at the start of iteration $j+1$, a will contain the sum of numbers in $A[0:j+1]$.
Thus the invariant holds.
Termination
When the *for* loop terminates $i = (n-1)+1 = n$.
Now the invariant states a holds the sum of all values in $A[0:n] = A$.
This is the exact value that the algorithm should return.
Thus the invariant holds.

Theorem 2.1 - *Floyd's Invariant Principle*
If P is a preserved invariant and is true for the base case, then P is true for all states.

1.3 Asymptotic Analysis

Remark 3.1 - *Motivation*
By performing asymptotic analysis on an algorithm we can better understand how it behaves as its input scales.
Algorithms with lower complexity perform better as their input scales up.

Definition 3.1 - *Asymptotic Analysis*
Asymptotic Analysis analyses the limiting behaviour of a function or algorithm as its input scales.
Asymptotic Analysis is often used to analyse the run-time & space requirements of an algorithm, other properties that can be analysed include cost & hardware requirement.
You can perform *Asymptotic Analysis* on the best, worst and average cases for each of these properties.

Definition 3.2 - *Big-Oh Notation*
Big-Oh Notation is used to classify functions by the upper bound of their growth.
Formal Definition

$$f(n) \in O(g(n)) \implies \exists\ n_0 > 0, c > 0\ st\ 0 \leq f(n) \leq c.g(n)\ \forall\ n > n_0.$$

Set Notation Definition

$$O(g(n)) = \{f(n)|\exists n_0 > 0, c > 0\ st\ 0 \leq f(n) \leq c.g(n)\ \forall\ n > n_0\}$$

Proposition 3.1 - *Order of Growth of Functions*
Here is the order of growth of select functions in *Big-Oh*.

$$O(1) \subset O(\log_2 n) \subset O(n) \subset O(n \log_2 n) \subset O(n^2) \subset O(2^n) \subset O(n!)$$

Remark 3.2 - *Occurrence of $f(n) \in O(log_2 n)$*
It is very rare to have functions with run-time complexity in $O(log_2 n)$ since reading and writing data is $O(n)$.

Definition 3.3 - *Big-Ω Notation*
Big-Ω Notation is used to classify functions by the lower bound of their growth.
Formal Definition

$$f(n) \in \Omega(g(n)) \implies \exists\ n_0 > 0, c > 0\ st\ 0 \leq c.g(n) \leq f(n) \forall\ n > n_0$$

Set Notation Definition

$$\Omega(g(n)) = \{f(n)|\exists\ n_0 > 0, c > 0\ st\ 0 \leq c.g(n) \leq f(n) \forall\ n > n_0\}$$

Definition 3.4 - *Big-Θ Notation*
Big-Θ Notation is used to classify functions by the upper *and* lower bound of their growth.
Set Notation Definition

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

Formal Definition

$$f(n) \in \theta(g(n)) \implies \exists\ n_0 > 0, c_1 > 0, c_2 > 0\ st\ 0 < c_1.g(n) \leq f(n) \leq c_2.g(n)\ \forall\ n \geq n_0$$

Proposition 3.2 - *Properties of Big-Θ*

i) If $f_1(n) \in \Theta(g(n))$ & $f_2(n) \in O(g(n)) \implies f_1(n) + f_2(n) \in \Theta(g(n))$

ii) If $a \in \mathbb{R}$ & $f(n) \in \Theta(g(n)) \implies af(n) \in \Theta(g(n))$.

iii) If $f_1 \in \Theta(g_1(n)), \dots, f_i \in \Theta(g_i(n))$ for $i \in \mathbb{N} \implies f_1(n)f_2(n)\dots f_i(n) \in \Theta(g_1(n)g_2(n)\dots g_i(n))$

Proof 3.1 - *Proposition 3.2*

i)

$$\begin{aligned}
&\text{Since } f_1(n) \in \Theta(g(n)) \text{ \& } f_2(n) \in O(g(n)) \\
&\quad \exists\ c_1, d_1, d_2 \in \mathbb{R}_{>0}\ st\ 0 < c_1 g(n) \leq f_1(n) \leq d_1 g(n) \text{ \& } 0 < f_2(n) \leq d_2 g(n) \\
&\implies 0 < c_1 g(n) \leq f_1(n) \leq f_1(n) + f_2(n) \leq (d_1 + d_2) g(n) \\
&\implies f_1(n) + f_2(n) \in \Theta(g(n))
\end{aligned}$$

ii)

$$\begin{aligned}
&\text{Since } f(n) \in \Theta(g(n)) \\
&\quad \exists\ a, c, d \in \mathbb{R}_{>0} \text{ \& } n_0 \in \mathbb{N}\ st\ cg(n) \leq f(n) \leq dg(n)\ \forall\ n \geq n_0 \\
&\implies acg(n) \leq af(n) \leq adg(n) \\
&\implies af(n) \in \Theta(g(n))
\end{aligned}$$

iii)

$$\begin{aligned}
&\text{Since } f_j(n) \in \Theta(g_j(n))\ \forall\ j \in \mathbb{N},\ 1 \leq j \leq n \\
&\quad \exists\ c_j, d_j \in \mathbb{R}_{>0} \text{ \& } n_0 \in \mathbb{N}\ st\ 0 < c_i g_i(n) \leq f_i(n) \leq d_i g_i(n)\ \forall\ n \geq n_0 \\
&\implies 0 < c_1 \dots c_i g_1(n) \dots g_i(n) \leq f_1(n) \dots f_i(n) \leq d_1 \dots d_i g_1(n) \dots g_i(n) \\
&\implies f_1(n) f_2(n) \dots f_i \in \Theta(g_1(n) g_2(n) \dots g_i(n))
\end{aligned}$$

Theorem 3.1 - *Master Theorem*
The *Master Theorem* is a multi-case formula for performing asymptotic analysis on *Divide-&-Conquer* recurrence relations.
By rearranging the recurrence relation to the form

$$T(n) = aT\left(\left[\frac{n}{b}\right]\right) + f(n) \quad a \geq 1, b > 1, f(n) > 0 \text{ for large } n$$

where [ ] means either *floor* or *ceil*.
By defining $p = \log_b a$, we can categorise $T(n)$ as

$$\begin{array}{lrcl}
1) & f(n) \in O(n^c),\ c < p & \implies & T(n) \in \Theta(n^p) \\
2) & f(n) \in \Theta(n^c),\ c = p & \implies & T(n) \in \Theta(n^p \log_2 n) \\
3) & f(n) \in \Omega(n^c),\ c > p \text{ \& conditions} & \implies & T(n) \in \Theta(f(n))
\end{array}$$

N.B. - The extra conditions in the *third case* as not specifically defined in this course, but are to do with the functions smoothness.

Example 3.1 - *Master Theorem*
Here we shall use the *Master Theorem* to analyse the following recurrence relations

i) $T(n) = 25T(n/5) + 4n^2$.
Here $a = 25,\ b = 5$ & $f(n) = 4n^2$.
Thus, $f(n) \in O(n^2) \implies c = 2$.
$\log_b a = \log_5 25 = 2 \equiv c$.
$T(n) \in \Theta(n^2 \log_2 n)$.

ii) $T(n) = 20T(n/5) + 4n$ Here $a = 20$ $b = 5$ & $f(n) = 4n$.
Thus, $f(n) \in O(n) \implies c = 1$.
$\log_b a = \log_5 20$.
Since $5^1 = 5 < 20$ then $\log_5 20 > 1 = c$.
$T(n) \in \Theta(n^{\log_5 20})$.

iii) $T(n) = 16T(n/2) + 2n^4$ Here $a = 16,\ b = 2$ & $f(n) = 2n^4$.
Thus, $f(n) \in O(n^4) \implies c = 4$.
$\log_b a = \log_2 16 = 4 \equiv c$.
$T(n) \in \Theta(n^4 \log_2 n)$.

Theorem 3.2 - *Akra-Bazzi Formula*
The *Akra-Bazzi Formula* is a single-case formula for performing asymptotic analysis on *Divide-&-Conquer* recurrence relations.
By rearranging the recurrence relation to the form

$$T(n) = \sum_{i=1}^{k} \left(a_i T\left([d_i n]\right)\right) + f(n) \quad \forall\ n \geq n_0$$

where $a_i > 0\ \forall\ i,\ 0 < b_i < 1\ \forall\ i$ & $\exists\ c \in \mathbb{R}\ st\ |f(n)| \in O(n^c)$ and [ ] means either *floor* or *ceil.*
Then

$$T(x) \in \Theta\left(x^p \left(1 + \int_{u=1}^{x} f(u) u^{-p-1} du\right)\right)$$

N.B. - The integral formula is non-examinable.

Proposition 3.3 - *Implications of Akra-Bazzi Formula*
From the *Akra-Bazzi Formula* we can perform asymptotic analysis on *Divide-&-Conquer* recurrence relations with the form

$$T(n) = \sum_{i=1}^{k} \left(a_i T([d_i n])\right) + \alpha n^c$$

where $\sum_{i=1}^{k} a_i \geq 1, a_i \in \mathbb{R}_{>0}\ \forall\ i, 0 < d_i < 1\ \forall\ i, \alpha \in \mathbb{R}_{>0}$ & $c \in \mathbb{R}$.
By setting p to be the solution of $\sum_{i=1}^{k} a_i(d_i^p) = 1$.
We can categorise $T(n)$ as

$$\begin{array}{lll} 1) & c < p & \implies T(n) \in \Theta(n^p) \\ 2) & c = p & \implies T(n) \in \Theta(n^p \log_2 n) \\ 3) & c > p & \implies T(n) \in \Theta(n^c) \end{array}$$

Proof 3.2 - *The Akra-Bazzi Formula is consistent with the Master Theorem*
Consider the preconditioned formulae of the *Master Theorem*

$$T(n) = aT\left(\left[\frac{n}{b}\right]\right) + \alpha n^c \quad a \geq 1, b > 1, \alpha \in \mathbb{R}_{>0}$$

and the *Akra-Bazzi Formula*

$$T(n) = \sum_{i=1}^{k} (a_i T([d_i n])) + \alpha n^c \quad k = 1, a_1 = a, d_1 = \frac{1}{b}$$

The preconditions for the Master Theorem ensure the conditions of the *Akra-Bazzi Formula.*
Set p to be the solution to $a_1 d_1^p = 1$.
Then $a\left(\frac{1}{b}\right)^p = 1 \implies p = \log_b a$.
By considering

1. If $\alpha n^c \in O(n^e)$ with $e < p$ then $c < p$, we statisfy *1)* for both.

2. If $\alpha n^c \in \Theta(n^e)$ with $e = p$ then $c = p$, we statisfy *2)* for both.

3. If $\alpha n^c \in \Omega(n^e)$ with $e > p$ then $c > p$, we statisfy *3)* for both.

Example 3.2 - *Akra-Bazzi Formula*
Here we shall use the *Akra-Bazzi Formula* to analyse the following recurrence relations

i) $T(n) = 2T(\lfloor n/4 \rfloor) + T(\lceil n/4 \rceil) + 15n^2$.
Here $k = 2,\ a_1 = 2,\ a_2 = 1,\ d_1 = \frac{1}{4},\ d_2 = \frac{1}{4},\ \alpha = 15, c = 2$.
Since $\sum_{i=1}^{2} a_i = 2 + 1 = 3 \geq 1$ & $0 < d_i < 1 \forall\ i$
this form obeys the conditions of the *Akra-Bazzi Formula.*

$$\begin{array}{rrcl} \text{Set} & \sum_{i=1}^{2} a_i d_i^p & = & 1 \\ \implies & 2\left(\frac{1}{4}\right)^p + \left(\frac{1}{4}\right)^p & = & 1 \\ \implies & \left(\frac{1}{4}\right)^p & = & \frac{1}{3} \\ \implies & p < 1 < 2 & = & c \end{array}$$

Thus $T(n) \in \Theta(n^2)$.

ii) $T(n) = \frac{1}{2}T(\lfloor n/2 \rfloor) + \frac{1}{3}T(\lfloor n/3 \rfloor) + \frac{1}{6}T(\lceil n/4 \rceil) + 6$.
Here $k = 3,\ a_1 = \frac{1}{2},\ a_2 = \frac{1}{3},\ a_3 = \frac{1}{6},\ d_1 = \frac{1}{2},\ d_2 = \frac{1}{3},\ d_3 = \frac{1}{6},\ \alpha = 6, c = 0$.
Since $\sum_{i=1}^{3} a_i = \frac{1}{2} + \frac{1}{3} + \frac{1}{6} = 1 \geq 1$ & $0 < d_i < 1 \forall\ i$
this form obeys the conditions of the *Akra-Bazzi Formula.*

$$\begin{array}{rrcl} \text{Set} & \sum_{i=1}^{3} a_i d_i^p & = & 1 \\ \implies & \frac{1}{2}\left(\frac{1}{3}\right)^p + \frac{1}{3}\left(\frac{1}{3}\right)^p + \frac{1}{6}\left(\frac{1}{6}\right)^p & = & 1 \\ \implies & p = 0 & = & c \end{array}$$

Thus $T(n) \in \Theta(n^0 \log_2 n) \equiv T(n) \in \Theta(log_2 n)$.

1.4 Recurrence

Remark 4.1 - *Solving Recurrences*
When solving recurrences we can use the following general processes:

i) Guess the general form of the solution,

ii) Use induction with this solution to check if the form is valid and to find values for variables.

Proposition 4.1 - *Recommended General Solutions for Recurrence Relations*
The following are general solutions for recurrence relations that take a defined form.

T(n)	General Solution
$c + T(n-1)$	$An + b$
$aT(n-1) + c$	$a^n A + B$
$T(n-1) + cn$	$An^2 + Bn + C$
$T(\frac{n}{b}) + cn$	$An \log_b n + Cn$
$aT(n-2) + bT(n-1)$	r^n

Example 4.1 - *Solving Recurrence Relation*
Here we solve the recurrence relation $f(n) = 2f(n-1) + 3f(n-2) + 1000$.
Let the initial conditions be $f(0) = 0$ & $f(1) = 1000$.
Assume general solution $f(n) = \alpha\lambda_1^n + \beta\lambda_2^n + \gamma$.

$$\begin{array}{rrcl}
\text{Set} & \lambda^2 & = & 2\lambda + 3 \\
 & 0 & = & \lambda^2 - 2\lambda - 3 \\
 & & = & (\lambda-3)(\lambda+1) \\
\implies & \lambda_1 = -1 & \& & \lambda_2 = 3 \\
\text{Set} & f(n) & = & \alpha(-1)^n + \beta 3^n + \gamma \\
\text{Since} & f(n) & = & 2f(n-1) + 3f(n-2) + 1000 \\
 & & = & 2[\alpha(-1)^{n-1} + \beta 3^{n-1} + \gamma] + 3[\alpha(-1)^{n-2} + \beta 3^{n-2} + \gamma] + 1000 \\
 & & = & 2\alpha(-1)^{n-1} + 2\beta 3^{n-1} + 2\gamma + 3\alpha(-1)^{n-2} + 3\beta 3^{n-2} + \gamma + 1000 \\
 & & = & 2\alpha(-1)^{n-1} + 2\beta 3^{n-1} + 2\gamma - 3\alpha(-1)^{n-1} + \beta 3^{n-1} + \gamma + 1000 \\
 & & = & -\alpha(-1)^{n-1} + \beta 3^n + 5\gamma + 1000 \\
 & & = & \alpha(-1)^n + \beta 3^n + 5\gamma + 1000 \\
 & & = & f(n) + 4\gamma + 1000 \\
\implies & 0 & = & 4\gamma + 1000 \\
\implies & \gamma & = & -250
\end{array}$$

$$\begin{array}{rrcl}
 & \text{From Conditions} & & \\
 & f(0) & = & 0 \\
\implies & \alpha + \beta - 250 & = & 0 \\
\implies & \beta & = & 250 - \alpha \\
 & f(1) & = & 1000 \\
\implies & -\alpha + 3\beta - 250 & = & 1000 \\
\implies & -\alpha + 3(250 - \alpha) & = & 1250 \\
\implies & -4\alpha & = & 500 \\
\implies & \alpha & = & -125 \\
\implies & \beta & = & 250 - (-125) \\
 & & = & 375 \\
 & & & \\
\implies & f(n) & = & -125(-1)^n + 375(3^n) - 250 \\
 & & = & 125(-1)^{n+1} + 275(3^n) - 250
\end{array}$$

1.5 P & NP

Remark 5.1 - *Motivation*
Most of this course has been on efficient algorithms & data structures.
Here we show that there may not always be an efficient algorithm for solving a problem.
N.B. - An algorithm is said to be efficient if it has polynomial run-time complexity.

Definition 5.1 - *P*
P is a class of problems.
P is the set of decision problem[1] families with polynomial run-time complexity.

Definition 5.2 - *Reduction*
A *Reduction* maps a problem family A to another problem family B, in polynomial run-time complexity.
The mapping is such that, A answers *yes* $\iff$ B answers *yes*.

[1]See **Reference:Definition 1.1**

Example 5.1 - *Reduction*
Consider the following decision problems about a graph G.

- *INDEPENDENT SETS* - Is there a set of k vertices in G such that no two vertices in the set are linked by an edge?.
- *CLIQUE* - Is there a set of K vertices in G such that every pair of vertices in the set is linked by an edge?

Here, *INDEPENDENT SETS(G,k)* answers *yes* $\iff$ *CLIQUE(G',k)* answers *yes.*
Where G' is the compliment of G.
N.B. - We can map G to G' in polynomial time.

Definition 5.3 - *Notation for Reduction*
If we have an algorithm with polynomial run-time complexity that, given any instance of decision problem family A, transforms it into an instance of decistion problem family B such that the answer produced by A is *yes* $\iff$ the answer produced by B is *yes.*
Then we say *"There is a polynomial-time reduction of A to B".*
More briefly *"A reduces to B".*
Or denoted $A \leq_P B$.
N.B. - In **Example 5.1** we have that *INDEPENDENT SETS* $\leq_P$ *CLIQUE.*

Remark 5.2 - *Solving Reductions*
If $A \leq_P B$ and we can solve B then we can use this to solve A.

Proposition 5.1 - *Properties of Reductions*

i) If $A \in P$ & $B \leq_P A$ then $B \in P$.
ii) For any decision problem family A, there is a polynomial reduction of A to A.
Thus, $A \leq_P A$.
iii) If $A \leq_P B$ & $B \leq_P C \implies A \leq_P C$.

Proposition 5.2 - *Verifying Solutions*
There are many problems that we don't know how to solve efficiently.
However, for some of these problems if someone claims the answer is *yes* and gives us evidence we can verify the claim efficiently.

Definition 5.4 - *NP*
NP is a class of problems.
NP is the set of problems for which, if the answer to an input is *yes*, we can verify it efficiently.
N.B. - *NP* stands for *Nondeterministic Polynomial.*

Remark 5.3 - *P & NP*
Let $A \in P$.
We can efficiently verify a claim that an input to A results in the answer yes by solving A. Thus $A \in P \implies A \in NP$
Further $P \subseteq NP$.
N.B. - Whether $P = NP$ is a *Millennium Prize Problem.*

Definition 5.5 - *NP-Hard*
$A \in NP - Hard \implies \forall\ B \in NP,\ B \leq_P A.$
This means A is at least as hard as the hardest problems in NP.

Proposition 5.3 - *Proving P=NP*
From these definitions there are several avenues that can be followed to try & prove $N = NP$.

- If there is a polynomial-time algorithm for a problem family in *NP-Hard* then $P = NP$.
- If there is a polynomial-time algorithm to solve just one of the *NP-Complete* problem families then $P = NP$.
- If you can prove that for just one *NP-Complete* problem family there is no efficient algorithm to solve it, then $P \neq NP$.

Definition 5.6 - *NP-Complete*
$A \in NP - Complete \implies A \in NP - Hard \cap NP.$

Example 5.2 - *NP-Compete Problems*
The following are NP-Complete problem families

- Timetable Scheduling;
- Finding Hamiltonian Circuits of Graphs;
- Solving an $n \times n$ Sudoku

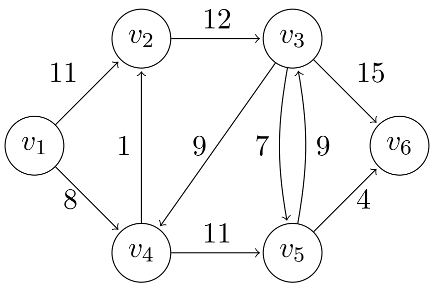
Proposition 5.4 - *Proving NP-Completeness by Reduction*
Let $A \in NP - Complete$ & $B \in NP$.
Then you can show that $A \leq_P B$.
Since $A \in NP - Hard$ then $\forall\ C \in NP,\ C \leq_P A$.
We have that $\forall\ C \in NP$ we have $C \leq_P A$ & $A \leq_P B$.
Thus $\forall\ C \in NP,\ C \leq_P B \implies B \in NP - Hard.$
Since $B \in NP$ & $B \in NP - Hard \implies NP - Complete.$

2 Data Structures

2.1 Graphs

Definition 1.1 - *Graph*
A *Graph* is a data structure made up of edges and vertices.
Vertices represent states of a system & edges show connections between two vertices.
Edges can be directed or undirected, they can also hold weightings.

Example 1.1 - *Graph*



Definition 1.2 - *Paths*
A *Path* in a graph is a sequence of nodes which are connected by a sequence of edges.
*Path*s can be further categorised as

- A *Trail* is a *Path* where every vertex within it is only met once.
- A *Walk* is a *Path* where every vertex & edge within it are only traversed once.
- A *Cycle* is a *Trail* which starts and ends on the same vertex.
- A *Simple Path* is a *Path* which has no repeating edges.

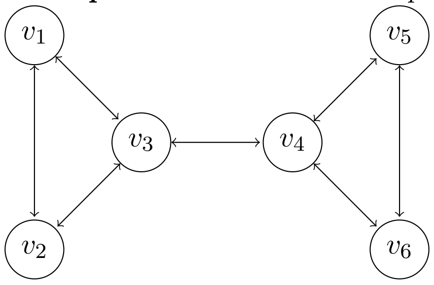
Example 1.2 - *Paths*
Consider the graph in **Example 1.1**.

- $v_1 \to v_2 \to v_3 \to v_45$ is a *Simple Path*, a *Trail* & a *Path.*
- $v_2 \to v_3 \to v_4 \to v_2$ is a *Cycle.*

Definition 1.3 - *Connected Graph*
A *Connected Graph* is a graph in which there exists at least one path between each pair of vertices.
If there exists one pair of edges which a path cannot be formed between, then the graph is said to be *disconnected.*

Definition 1.4 - *Bridge*
A *Bridge* is an edge of a connected graph which, if it was deleted, would leave the graph disconnected.

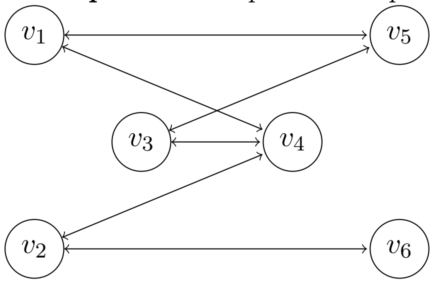
Example 1.3 - *Connected Graph & Bridge Edge*



N.B. - The edge (v_3, v_4) is a bridge.

Definition 1.5 - *Bipartite Graphs*
Bipartite Graph is a graph where its vertices can be divided into independent disjoint sets such that every edge connects a vertex from one set, to a vertex in the other.

Example 1.4 - *Bipartite Graph*



This graph can be split into $[v_1, v_2, v_3]$ & $[v_4, v_5, v_6]$ to be a bipartite graph.

Definition 1.6 - *Node Degree*
The *Degree* of a node in an *undirected graph* is the number of edges connected to that node.
The *Degree* of a node in an *directed graph* is the number of edges into the node less the number of edges leaving it.

Proposition 1.1 - *Representing Graphs as Matrices*
A directed graph with $|V|$ vertices can be represented by a $|V| \times |V|$ matrix, A.[1]
Where the value of A_{ij} gives the weight of the edge from $i \to j$.
If the value of A_{ij} is 0 then there is no edge between vertex i & vertex j.
If the graph is unweighted then the value of A_{ij} doesn't matter as long as it is not zero.

Proposition 1.2 - *Representing Graphs as Adjacency Lists*
Graphs can be represented by an *Adjacency List.*[2]
An *Adjacency List* can be represented by a map from a vertex to a linked list.
The linked list stores the vertices that the vertex has an edge to and the weight of that edge as tuples.

Remark 1.1 - *Comparison of Graph Representations*
Let $|V|$ be the number of vertices & $|E|$ be the number of edges in a Graph.

	Space	Is there an edge $i \to j$	Get all edges leaving i
Matrix	$\Theta(\lvert V\rvert^2)$	$O(1)$ Time	$O(\lvert V\rvert)$ Time
Adjacency List	$\Theta(\lvert V\rvert + \lvert E\rvert)$	O($degree(i)$) Time	O($degree(i)$) Time

N.B. - Adjacency lists take up a lot of space when loaded into memory.

[1]See `programs\MatrixGraph.java`
[2]See `programs\LinkedGraph.java`

Definition 1.7 - *Tree*
A *Tree* is an undirected graph where each node can be connected by exactly one path.

Definition 1.8 - *Binary Tree*
A *Binary Tree* is a tree where each node has either 0, 1 or 2 children.

Example 1.5 - *Binary Tree*

```
        2
      /   \
   7         5
  / \        |
2    6       9
    / \
   5 11
```

Definition 1.9 - *Balance*
A tree is said to be *Balanced* if the heights of the left & right subtrees of all vertices differ by no more than one.

Definition 1.10 - *Binary Tree Rotation*
A *Rotation* is an operation on a binary tree that changes the structures of the tree without affect the order of elements.
They are used to balance trees so that they are as short as possible and thus quick to search.
N.B. - The vertex we balance around is called the *pivot*.

Definition 1.11 - *Left Rotation*
A *Left Rotation* can be used to balance a binary tree if the left subtree of a given node causes an imbalance at that node.
A *Left Rotation* can be performed by the following process

1. Remove the pivot's left subtree.
2. Make the parent, and its left subtree, the pivot's righleft child.
3. Make the removed subtree the right child of the old parent.

Example 1.6 - *Left Rotation*
Here we shall perform a left rotation on the following tree.

```
   2
  / \
7    5
    / \
   4  6
```

To do so we must pivot around 5.
Left child.

```
4
```

Parent and its left child.

```
2
|
7
```

Pivot and its right child.

```
5
|
6
```

Make parent the pivot's right child.

```
   5
  / \
 2   6
 |
 7
```

Make the removed child the parent's left child.

```
    5
   / \
  2   6
 / \
 7 4
```

Definition 1.12 - *Right Rotation*
A *Right Rotation* is used to balance a binary tree if the left subtree of a node is taller than the right.
A *Right Rotation* can be performed by the following process

1. Remove the pivot's right subtree.
2. Make the parent, and its right subtree, the pivot's right child.
3. Make the removed subtree the left child of the old parent.

Example 1.7 - *Right Rotation*
Here we shall perform a right rotation on the following tree.

```
    2
   / \
  7   5
 / \
 4 6
```

To do so we must pivot around 7.
Right child.

```
 6
```

Parent and its left child.

```
 2
 |
 5
```

Pivot and its left child.

```
 7
 |
 4
```

Make parent the pivot's right child.

```
   7
  / \
 4   2
     |
     5
```

Make the removed child the parent's left child.

```
   7
  / \
 4   2
    / \
    6 5
```

Definition 1.13 - *Complete Binary Tree*
A *Complete Binary Tree* is a binary tree where every layer is full, except for the bottom layer.

Example 1.8 - *Complete Binary Tree*

```
    2
   / \
  7   5
 / \   |
2 6   9
```

Definition 1.14 - *Binary Heap*
A *Binary Heap* is a complete binary tree which satisfies a heap ordering property.
There are two heap ordering properties

i) *Min-Heap Property* - The value of each node is greater than, or equal to, its parent.

ii) *Max-Heap Property* - The value of each node is less than, or equal to, its parent.

Example 1.9 - *Binary Heap*

```
    9           1
   / \         / \
  6   3       2   7
 / \  / \    / \  / \
5 4 2 1     5 4 8 9
```

Remark 1.2 - *Implementing Binary Heap - Array*
[1] A *Binary Heap* can be implemented using an array by storing the values layer by layer.
This means the n^{th} layer starts at index $2^n - 1$.
For element i in the list we can get the index of its parent & children using

- $Parent(i) = \lfloor \frac{i}{2} \rfloor$.
- $Left - Child(i) = 2i + 1$
- $Right - Child(i) = 2i + 2$

Remark 1.3 - *Finding Elements in Binary Heap*
To store elements in a binary heap, rather than just positive integers, create a lookup table maps a positive integer to a pointer to the element.
This assigned value is the element's ID.
To find the element with ID i, look up the i^{th} element of the lookup table.
This will be a pointer to that element.

2.2 Priority Queue

Definition 2.1 - *Priority Queue*
A *Priority Queue* is an abstract data type which stores a set of distinct elements.
Each element has an associated value, called its *key*.
All *Priority Queue*s can perform the following operations.

i) $INSERT(x, k)$ - Inserts new element x and sets $x.key = k$.

ii) $DESCREASE - KEY(x, k)$ - Decrease a $x.key$ to equal k, where $k < x.key$.

iii) $EXTRACT-MIN()$ - Removes & returns the element with the smallest value key. When multiple elements have the lowest value, one is returned arbitrarily.

[1] See `programs\BinaryHeapPriorityQueue.java`

Proposition 2.1 - *Implementing Priority Queue - Unsorted Linked-List*
An *Unsorted Linked-List* can be used to implement a priority queue.
The operations of a *Priority Queue* can be emulated by

i) $INSERT(x, k) \in O(1)$.
Element is placed at the start of the list.

ii) $DESCREASE - KEY(x, k) \in O(n)$.
Increment through the list until x is found.

iii) $EXTRACT - MIN() \in O(n)$.
Increment through the list until x is found.

N.B. - A coded example of this can be seen in `programs/UnsortedLinkedPriorityQueue.java`

Proposition 2.2 - *Implementing Priority Queue - Sorted Linked-List*
A *Sorted Linked-List* can be used to implement a priority queue.
The operations of a *Priority Queue* can be emulated by

i) $INSERT(x, k) \in O(n)$.
Incrementing through the list until you find an element whose key is greater than k.
Insert x before this element.

ii) $DESCREASE - KEY(x, k) \in O(n)$.
Increment through the list until you find x, decrease its key and move it forward until you encounter an element whose key is less less than k.
Place decremented element after this element.

iii) $EXTRACT - MIN() \in O(1)$.
Lowest value element is always are front of list.

N.B. - A coded example of this can be seen in `programs\SortedLinkedPriorityQueue.java`

Proposition 2.3 - *Implementing Priority Queue - Binary Heap*[1]
A *Minimum Binary Heap* can me used to implement a priority queue.
The height of the heap is $\lceil \log_2 n \rceil$, so this is the maximum number of elements which need to be checked in any operation.
Reinstating the heap property of a binary heap is $\in O(\log_2 n)$.
The operations of a *Priority Queue* can be emulated by

i) $INSERT(x, k) \in O(\log_2 n)$.
Add the element to the bottom of the heap.
Compare k to the key of the new element's parent, if k is less then swap the nodes.
Repeat this until the parent's key is less than k or you have reached the top of the list.

ii) $DESCREASE - KEY(x, k) \in O(\log_2 n)$.
Find x in the heap and update its key.
Compare k to the key of the new element's parent, if k is less then swap the nodes.
Repeat this until the parent's key is less than k or you have reached the top of the list.

iii) $EXTRACT - MIN() \in O(\log_2 n)$.
Remove the top element from the heap.
Move the last element of the heap to be the root.
If its left child has a lower key then swap them, else if the right child has a lower key then swap them.
Repeat until both children have a greater key or you have reached the bottom of the list.

[1]See `programs\BinaryHeapPriorityQueue.java`

Remark 2.1 - *Summary of Priority Queue Implementations*
Here is a summary of the space complexity and run-time complexity for the operations required to implement a priority queue for different data structures.
Let N to be the maximum key for element.

	INSERT	DECREASE-KEY	EXTRACT-MIN	SPACE
Unsorted Linked-List	O(1)	O(n)	O(n)	O(n)
Sorted Linked-List	O(n)	O(n)	O(1)	O(n)
Binary Heap	O($log_2 n$)	O($log_2 n$)	O($log_2 n$)	O(N)

N.B. - Fibonacci heaps are better than binary heaps, however they are complex and generally not practical. (They are not in this course).

2.3 Disjoint Sets

Definition 3.1 - *Disjoint Sets*
Disjoint Sets is an abstract data structure.
It emulates a collection of sets of elements that have no common values between or within them.
Each set has a unique identifier, called its *identity.*
The *identity* can be implemented in any way.
All *Disjoint Sets* can perform the following operations

i) $MAKE-SET(x)$ - Makes a new set containing only x.

ii) $UNION(x, y)$ - Merges the set containing element x with the set containing element y.

iii) $FIND-SET(x)$ - Returns the *identity* of the set contain x.

Theorem 3.1 - *Uniqueness of FINDSET(x)*
$FIND-SET(x) \equiv FIND-SET(y)$ iff x and y are in the same set.

Remark 3.1 - *Operations of Disjoint Sets enforce Disjointedness*
Two sets cannot have a shared element since first you must call $MAKE-SET(x)$ which will not run if x is already in a set.
There is no other way to add elements to disjoint sets.

Proposition 3.1 - *Implementation of Disjoint Sets - Reverse Tree*
A collection of *Reverse Trees* can be used to implement disjoint sets.
Reverse trees can be stored in an array of fixed length where the index of an element gives the value and the item in that index is a pointer to its parent.
You need unique values to distinguish parents & unused values.
The root of each tree is the identity of that set.
Children point to parents, but parents do not point to children.
The operations of *Disjoint Sets* can be emulated by

i) $MAKE-SET(x) \in O(1)$.
Expand array to accommodate x.
Set x to be a parent.

ii) $UNION(x, y) \in O(tree-height)$
Set $r_x = FIND-SET(x)$ & $r_y = FIND-SET(y)$.
Point r_y to r_x.

iii) $FIND - SET(x) \in O(tree - height)$.
Find the pointer in index x.
Follow its pointer to the next element.
Repeat this until the pointer indicates the element is a parent.
Return the index of this pointer

N.B. - A coded example of this can be seen in `programs/ReverseTreeDisjointSets.java`.

Proposition 3.2 - *Height of Reverse Tree*
In the worst case implementation of $UNION(x, y)$ the tree height will increase by one every time, causing all operations to run in $O(n)$.
This can improved by balancing the trees.
One way is to store the heights of the trees and always making the shorter tree the child of the other.
By doing this, the height of the resulting tree only increases, by one, if the two previous trees were the same height.
N.B. - This is the version implemented in `programs/ReverseTreeDisjointSets.java`.

Proof 3.1 - *The height of the tallest tree is $log_2 n$*
Here we shall prove that by using the previously described process the tallest resulting trees will have height $log_2 n$.
This is a proof by contradiction.
We begin by proving that any tree of height h contains at least 2^h nodes.
This is a proof by induction.
Base Case
Set $i = 0$.
A tree of height 0 is a single node. $2^i = 2^0 = 1$.
The claim holds.
Inductive Step
Assume that every tree of height $i - 1$ contains at least 2^{i-1} nodes.
$UNION(x, y)$ only produces a tree of height i when passed two trees of height $i - 1$.
Therefore, a tree of height i contains $2^{i-1} + 2^{i-1} = 2.2^{i-1} = 2^i$ nodes.
Now assume, for contradiction, that there is a tree of height $h \geq \log_2 n + 1$.
This tree contains at least $2^{log_2 n+1}$, which is greater than n, nodes and each node represents a distinct element.
This is a contradiction because the elements are members of the set $\{1, 2, \dots, n\}$ so $h \leq \log_2 n$.

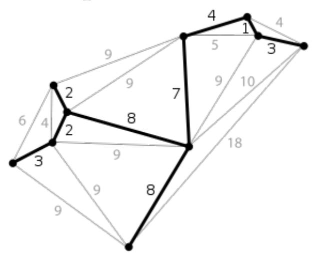
2.4 Spanning Trees

Definition 4.1 - *Spanning Tree*
A *Spanning Tree* is a sub-graph of a connected, undirected graph that contains every element of the graph and has no cycles.

Definition 4.2 - *Weight of Spanning Tree*
The *weight* of a spanning tree is the sum of the weights of its edges.

Definition 4.3 - *Minimum Spanning Tree*
A *Minimum Spanning Tree* of a given graph is the spanning tree with the lowest weight.

Example 4.1 - *Minimum Spanning Tree*



Remark 4.1 - *Finding Minimum Spanning Trees*
Kruskal's Algorithm can be used to find minimum spanning trees[1].

2.5 Dynamic Search Structures

Definition 5.1 - *Dynamic Search Structures*
A *Dynamic Search Structure* stores a set of elements.
Each element x has a unique key, $x.key$.
All *Dynamic Search Structure*s can perform the following operations

i) $INSERT(x, k)$ - Inserts element x with key k.

ii) $FIND(k)$ - Returns the unique element x where $x.key \equiv k$.

iii) $DELETE(k)$ - Deletes the unique element x where $x.key \equiv k$.

Remark 5.1 - *Addition Operations of Dynamic Search Structures*
The following are additional operations which are useful for *Dynamic Search Structures* to support

i) $PREDECESSOR(k)$ - returns the unique element x with the largest key st $x.key < k$.

ii) $RANGE - FIND(k_1, k_2)$ - returns every element x where $k_1 \leq x.key \leq k_2$.

Remark 5.2 - *Usefulness of PREDECESSOR*
$PREDECESSOR$ is used when deleting elements from structures with heap properties as replacing an element with its predecessor ensures the heap property is maintained.

Proposition 5.1 - *Implementing Dynamic Search Structure - Unsorted Linked List*
An *unsorted linked list* is a valid data structure to implement as a dynamic search structure.
The run-time complexity for the operations is

i) $INSERT \in O(1)$.

ii) $FIND \in O(n)$.

iii) $DELETE \in O(n)$.

where n is the number of elements in the linked list.

[1]See **Subsection 3.9**

Proposition 5.2 - *Implementing Dynamic Search Structure - Binary Search Tree*
A *Binary Search Tree* is a valid data structure to implement as a dynamic search structure.[1]
The operations can be implemented as follows

i) $INSERT(x, k) \in O(h)$.
Traverse down the tree comparing node keys to k.
If a node's key $= k$ throw an error saying the key is not unique.
If a node's key $> k$ follow its left subtree.
If a node's key $< k$ follow its right subtree.
If a node does not have a required subtree, insert the element as a leaf there.

ii) $FIND(k) \in O(h)$.
Traverse down the tree comparing node keys to k.
If a node's key $= k$ return the associated element.
If a node's key $> k$ follow its left subtree.
If a node's key $< k$ follow its right subtree.
Add new key and element as a child to leaf.

iii) $DELETE(k) \in O(h)$.
Traverse down the tree, if you reach a leaf that is not the desired key return FAIL.
When key is found

  (a) If node has no children, delete node.
  (b) If node only has left subtree, delete node & replace with left child.
  (c) If node only has right subtree, delete node & replace with right child.
  (d) If node has both children, find its predecessor and replace node with it.

where h is the height of the tree.
N.B. - $h \geq \log_2 n$.

Proposition 5.3 - *Implementing Dynamic Search Structure - 2-3-4 Tree*
A *2-3-4 Tree*[2] is a valid data structure to implement as a dynamic search structure.[3]
The operations can be implemented as follows

i) $INSERT(x, k) \in O(h) \equiv O(\log_2 n)4$.
Traverse down the tree, comparing node keys to k.
Whenever you encounter a 4 *node* perform $SPLIT$ on it.
If a node contains k as a key, throw an error saying k is not unique.
Otherwise, follow the subtree which k is bounded on.
If you reach a leaf, insert the value by

  (a) If leaf is 2 *node* convert to 3 *node* with the new key, k.
  (b) If leaf is 3 *node* convert to 4 *node* with the new key, k.

ii) $FIND(k) \in O(h) \equiv O(\log_2 n)$.
Traverse down the tree, comparing node keys to k.
Whenever you encounter a 4 *node* perform $SPLIT$ on it.
If a node contains k as a key, return it.
Otherwise, follow the subtree which k is bounded on.
If you reach a node without the appropriate child, return FAIL.

[1]See `programs\BinarySearchTree.java`
[2]See **Definition 6.3**
[3]See `programs\Tree234.java`

iii) $DELETE(k) \in O(h) \equiv O(\log_2 n)$.
Traverse down the tree, comparing node keys to k.
Whenever you encounter a 2 *node* perform $FUSE$ or $TRANSFER$ on it.
If a node does not contain k as a key, follow the subtree which k is bounded on.
If you reach a leaf and it doesn't contain k as a key, throw an error saying key is unused.
Otherwise

(a) Search for key k using $FIND(k)$. Perform $FUSE$ or $TRANSFER$ on all 2 *node*s met here.
(b) If node is a leaf && a 3 *node* remove key and downgrade to 2 *node*.
(c) If node is a leaf && a 4 *node* remove key and downgrade to 3 *node*.
(d) If node is **not** a leaf:
  i. Use $PREDECESSOR(k)$ to find a leaf k'.
  ii. Perform $DELETE(k')$.
  iii. Overwrite k with k'.

N.B. - By performing $SPLIT$ on all met 4 *node*s, we will never end up trying to add the new key to a 4 *node*.
N.B. - By performing $FUSE$ or $TRANSFER$ on all met 2 *node*s, we will never end up trying to delete a key from a 2 *node*.

Proposition 5.4 - *Implementing Dynamic Search Structure - Skip List*
A *Skip List*[1] is a valid data structure to implement as a dynamic search structure.[2]
The operations can be implemented as follows

1. $INSERT(x, k) \in O(\log_2 n)$ on average.[3]
   Start at the first element of the top level.
   Move along the level until you find $k' > k$.
   Move down a level at the key directly before k'.
   Repeat moving along and down until you reach the bottom level.
   Move along bottom level until you find $k' > k$.
   Insert (x, k) directly before k'. Repeatedly flip a coin
   (a) HEADS - Insert (x, k) into the level above.
   (b) TAILS- stop.
2. $FIND(k) \in O(\log_2 n)$ on average.[4]
   Start at the first element of the top level.
   Move along level until you find $k' > k$.
   Move down a level at the key directly before k'.
   Repeat moving along and down until you reach the bottom level.
   Move along bottom level until you find $k' > k$.
   Move along the bottom level.
   If you find k return the associated element.
   If you find $k' > k$ return FAIL.

[1]See **Definition 7.2**
[2]See `programs\SkipList.java`
[3]See **Proposition 5.6**
[4]See **Proposition 5.6**

3. $DELETE(k) \in O(\log_2 n)$ on average.[1]
   Perform $FIND(k)$.
   If this fails, return FAIL.
   Otherwise, delete (x, k) from all layers.

Theorem 5.1 - *The Union Bound Theorem*
Let $E_1, E_2, \dots, E_n$ be events where the probabilty of E_j occuring is p_j.
Then the probability of at least one of $E_1, \dots, E_j$ occuring is at most $\sum_{i=1}^{j} p_i$.

Remark 5.3 - *Run-Time Complexity for Skip List Operations*
Skip Lists having a randomised algorithm for their construction.
The run-time complexity depends on the number of levels.
Thus run-time complexity for all their operations is given for the average case.

Proposition 5.5 - *Run-Time Complexity of FIND with a Skip List*
Since the number of levels is $O(\log_2 n)$ we can conclude that the number of times we move *down* the skip list is $O(\log_2 n)$.
For any given element, in any level, the probability of the element to its immediate right is also being the element to its immediate right in the bottom layer is $\frac{1}{2}$.
We can conclude that we are likely to move right $O(\log_2 n)$ times.
Since each movement has run-time complexity $\in O(1)$ we conclude that $FIND \in O(\log_2 n)$.

Remark 5.4 - *Comparing Implementations of Dynamic Search Structures*
Let n be the number of elements being stored.

	$INSERT(x, k)$	$DELETE(k)$	$FIND(k)$	Space
Unsorted Linked List	$O(1)$	$O(n)$	$O(n)$	$O(n)$
Binary Search Tree	$O(n)$	$O(n)$	$O(n)$	$O(n)$
2-3-4 Tree	$O(\log_2 n)$	$O(\log_2 n)$	$O(\log_2 n)$	$O(n)$
Red-Black Tree	$O(\log_2 n)$	$O(\log_2 n)$	$O(\log_2 n)$	$O(n)$
Skip List	$O(\log_2 n)$	$O(\log_2 n)$	$O(\log_2 n)$	$O(n)$

2.6 Self-Balancing Trees

Definition 6.1 - *Prefect Balance*
A tree has *Perfect Balance* if the length of every path from the root to any leaf is exactly the same as the height of the tree.

Definition 6.2 - *Self-Balancing Trees*
A *Self-Balancing Tree* automatic ensures perfect balance when inserting or deleting elements.

[1]See **Proposition 5.6**

Definition 6.3 - *2-3-4 Tree*
A *2-3-4 Tree* is a tree where all nodes have $0, 2, 3$ or 4 children only.
The nodes are named as

- 2 *node* which have 2 children & 1 key.
- 3 *node* which have 3 children & 2 keys.
- 2 *node* which have 4 children & 3 keys.

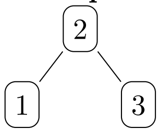
A *2-3-4 Tree* can implemented as a *Dynamic Search Structure.*[1]
N.B. - Each node has either the specified number of children or none.

Remark 6.1 - *Why 2-3-4?*
By seeing the efficiencies of a *2-3-4 Tree* we may consider why we don't use a *2-3-...-n Tree.*
This is because the more children a node has, the flatter the tree becomes and the more nodes we have to deal with at each level, rather than just moving down sub-trees.
It seems *2-3-4 Tree* maximises these efficiencies.

Remark 6.2 - *Height of* $2-3-4$ *Tree*
The height, h, of a $2-3-4$ tree is $\log_4 n \leq h \leq \log_2 n$.
This boundaries represent the case where all the nodes are 4 *nodes* and the case where all the nodes are 2 *nodes.*

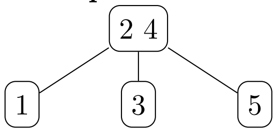
Definition 6.4 - *2 Node*
A 2 *node* has a single key and two sub-trees.
All the elements in the left sub-tree have a value less than the key,
and all those in the right sub-tree have a value greater than the key.

Example 6.1 - *2 Node*



Definition 6.5 - *3 Node*
A 3 *node* has two keys and three sub-trees.
Let x, y be the values of the keys st $x < y$.
All the elements in the left sub-tree have a value $< x$.
All the elements in the middle sub-tree have a value $> x$ AND $< y$.
All the elements in the right sub-tree have a values $> y$.

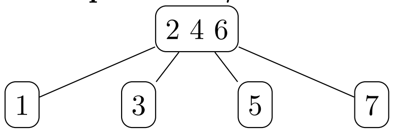
Example 6.2 - *3 Node*



[1]See `programs\Tree234.java`

Definition 6.6 - *4 node*
A 4 *node* has three keys and four sub-trees.
Let x, y, z be the values of the keys st $x < y < z$.
All the elements in the left subtree have a value $< x$.
All the elements in the middle-left subtree have a value $> x$ AND $< y$.
All the elements in the middle-right subtree have a value $> y$ AND $< z$.
All the elements in the right subtree have a value $> z$.

Example 6.3 - *4 Node*



Proposition 6.1 - *Inserting Elements into 2-3-4 Tree*
To insert a new element to a *2-3-4 Tree* traverse down the tree, following the correct path.
Whenever you encounter a 4 *node*, split it and continue traversing.
Once you reach a leaf insert the value into the node.

Proposition 6.2 - *Deleting Leaf from 2-3-4 Tree*
To delete a value from a leaf first traverse down the tree to the leaf.
If the leaf is a 2 *node* transfer or fuse it with a neighbour.
Remove the element as a key from the leaf.

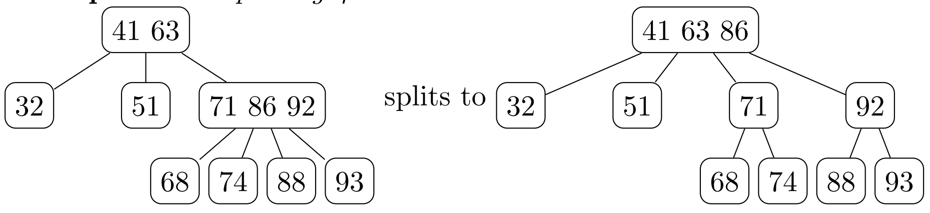
Proposition 6.3 - *Deleting from Node in 2-3-4 Tree*
To delete a value from a node first traverse to the node.
Find the predecessor to the value you are trying to delete.
Delete the predecessor from the tree.
Replace the value you are deleting with its predecessor.

Definition 6.7 - *Splitting 4 Nodes*
We can *split* any 4 *node* into two 2 *node*s, provided the 4 *node*'s parent is not a 4 *node* (otherwise the parent would have too many children).
Let n be a 4 *node* and p be its parent.
We can split n using the following process

1. Move the middle key of n to the appropriate place in p's keys.
2. Make a new 2 *node* using the lesser of the two remaining keys of n & the two leftmost subtrees of n.
3. Make another new 2 *node* using the remaining key & subtrees of n.
4. Make these two new 2 *node*s children of p, ensuring numerical order is maintained.

N.B. - Splitting a node does not affect the height of the tree.

Example 6.4 - *Splitting 4 Node*

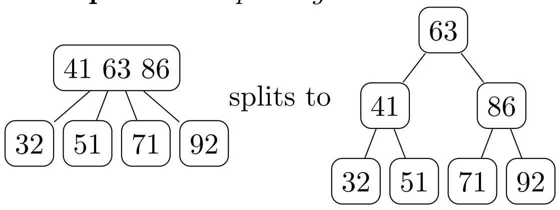


Definition 6.8 - *Splitting the Root*
Since the root doesn't have a parent we do not need to consider the restrictions on splitting it.
hen splitting a 4 *node* root we create a new 2 *node* root above it.
Let n be the root.
We can split n using the following process

1. Make a new 2 *node* with the least value of n and its two leftmost subtrees.
2. Make another new 2 *node* with the greatest value of n and its two rightmost subtrees.
3. Make a final 2 *node* with the middle value of n.
4. Make the other new nodes the children of the last, ensuring numerical order is maintained.

N.B. - Splitting the root increases the height of the tree by 1.
N.B. - This has run-time complexity $\in O(1)$.

Example 6.5 - *Splitting the Root*

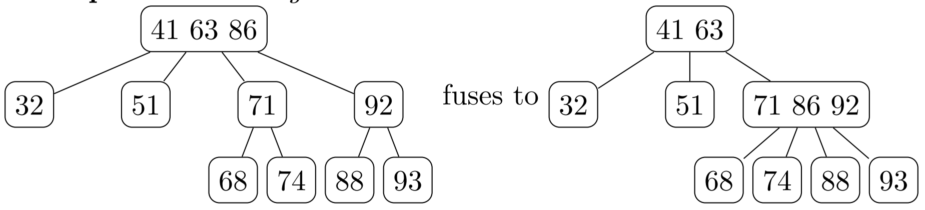


Definition 6.9 - *Fusing Two* 2 *nodes*
We can *fuse* two 2 *node* siblings into a single 4 *node*, provided their parent is not a 2 *node*.
Fusing runs in constant time.
Let n_1 and n_2 be 2 *node* siblings and p be their parent.
We can *fuse* n_1 & n_2 with $n_1 < n_2$ using the following process

1. Remove the key of p which forms the boundary between n_1 and n_2.
2. Create a new 4 *node* with the keys of n_1, n_2 and the key which was just removed from p, ensuring order is kept.
3. Make the sub-trees of n_1 the two leftmost sub-trees of the new node, and the sub-trees of n_2 as the two rightmost sub-trees of the new node.

N.B. - Fusing does not affect the height of the tree.

Example 6.6 - *Fusing Two* 2 *nodes*

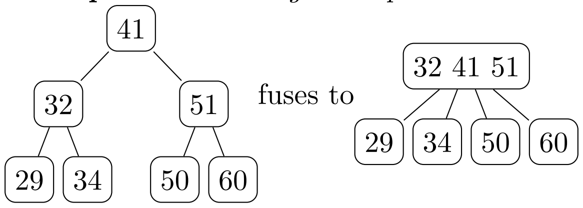


Definition 6.10 - *Fusing when parent is the Root*
We said you cannot fuse two nodes if their parent is a 2 *node*, an exception is made for when the parent is the root.
Here the root key is brought down to the two children.
Let n_1 and n_2 be the two 2 *nodes* siblings, whose parent is the root.
We can fuse n_1 & n_2 with $n_1 < n_2$ using the following process

1. Create a new 3 *node* using the keys of $n_1,\ n_2$ & the root.
2. Make the children of n_1 the leftmost children of the new node.
3. Make the children of n_2 the rightmost children of the new node.

N.B. - This reduces the height of the tree by 1.

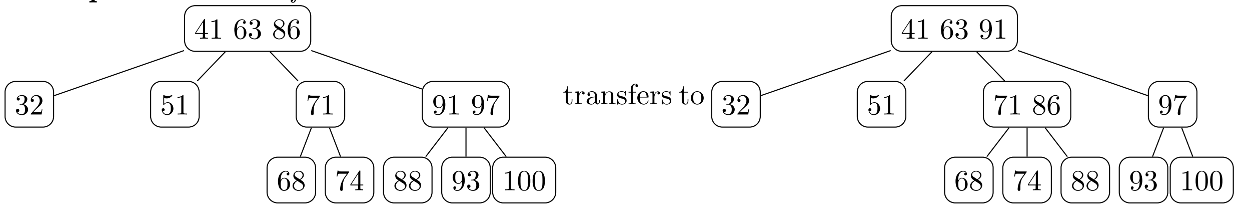
Example 6.7 - *Fusing when parent is the Root*



Definition 6.11 - *Transfer a* 2 *Node and a* 3 *Node*
If a 2 *node* is a sibling with a 3 *node* we can transfer between the two. Let n_2 be a 2 *node* and n_3 be a 3 *node* which are siblings and p be their parent.
We can transfer between n_2 & n_3

1. Identify the key of p which forms a boundary between n_2 and n_3.
2. Move this key to n_2, ensuring numerical order is maintained.
3. Move the key of n_3 which is closest in value to the keys of n_2 to p, ensuring order is kept.
4. Move the sub-tree of n_3 which is closest to n_2 to n_2, ensuring numerical order is maintained.

Example 6.8 - *Transfer a* 2 *Node and a* 3 *Node*



Remark 6.3 - *Implementing 2-3-4 Tree*
Since it is complicated to implement *2-3-4 Trees* in code, coders often use *Red-Black Trees* instead.

2.7 Skip Lists

Definition 7.1 - *Multi-Level Linked Lists*
A *Multi-Level Linked List* is a linked list which has several linked list above it.
Each of these linked lists is called a layer.
Each layers contains a subset of the elements of the list below it.
All layers contain the first and last elements of the linked list.
Each element links to its equivalent element in the list below.

Example 7.1 - *Multi-Level Linked List*

$$\begin{tikzcd}
1 \arrow[rr] \arrow[d] & & 5 \arrow[rrrr] \arrow[d] & & & & 25 \arrow[d] \\
1 \arrow[rr] \arrow[d] & & 5 \arrow[rr] \arrow[d] & & 16 \arrow[r] \arrow[d] & 18 \arrow[r] \arrow[d] & 25 \arrow[d] \\
1 \arrow[r] & 2 \arrow[r] & 5 \arrow[r] & 9 \arrow[r] & 16 \arrow[r] & 18 \arrow[r] & 25
\end{tikzcd}$$

Proposition 7.1 - *FIND(k) - Multi-Level Linked List*
Let k be the key we are trying to find.

i) Start at first element of top level.

ii) Move right along level until you find key $k' > k$.

iii) Move down a level at the key before k'.

iv) Repeat ii) & iii) until reach you kind k and are in bottom level.

Remark 7.1 - *Complexity of FIND(k) - Two-Level Linked List*
Let m be the number of keys in the top row & n be the number of elements in the bottom row.
If the m elements are evenly distributed $FIND(k) \in O(m + \frac{n}{m})$.
By setting $m = \sqrt{n}$ we get $FIND(k) \in O(m + \frac{n}{m}) = O(\sqrt{n})$.

Definition 7.2 - *Skip List*
A *Skip List* is a multi-level linked list which is constructed in a specific way.
The construction algorithm is a *randomised algorithm.*

i) Create linked list of all elements and place as bottom level.

ii) Add level above containing just the first and last element.

iii) For each of the other elements randomly toss a coin:

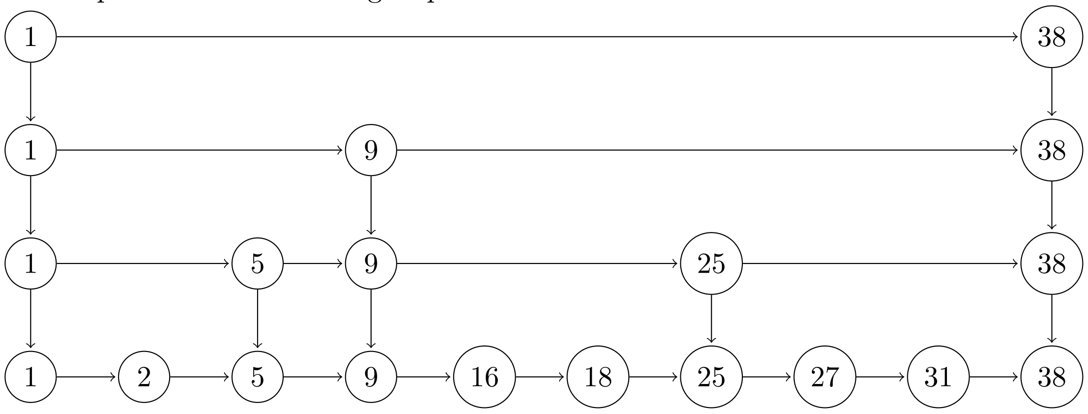
  (a) HEADS - Add element to new layer.
  (b) TAILS - Don't add element to new layer.

iv) Repeat ii) & iii) until you produce a layer containing just the first and last elements.

Example 7.2 - *Constructing Skip List*
Consider the following linked list and coin flips

$$\begin{tikzcd}
1 \arrow[rrr] \arrow[d] & & & T \arrow[rrrrrr] \arrow[d] & & & & & & 38 \arrow[d] \\
1 \arrow[rr] \arrow[d] & & T \arrow[r] \arrow[d] & H \arrow[rrr] \arrow[d] & & & T \arrow[rrr] \arrow[d] & & & 38 \arrow[d] \\
1 \arrow[r] \arrow[d] & T \arrow[r] \arrow[d] & H \arrow[r] \arrow[d] & H \arrow[r] \arrow[d] & T \arrow[r] \arrow[d] & T \arrow[r] \arrow[d] & H \arrow[r] \arrow[d] & T \arrow[r] \arrow[d] & T \arrow[r] \arrow[d] & 38 \arrow[d] \\
1 \arrow[r] & 2 \arrow[r] & 5 \arrow[r] & 9 \arrow[r] & 16 \arrow[r] & 18 \arrow[r] & 25 \arrow[r] & 27 \arrow[r] & 31 \arrow[r] & 38
\end{tikzcd}$$

Which produces the following skip list



Proposition 7.2 - *Number of Levels in Skip List*
An empty skip list contains only one level, and the only way to increase the number of levels is during $INSERT$.
The probability of increasing the number of levels if $\frac{1}{2}$ and the number of levels only every increases by 1.
Thus the probability of $INSERT$ adding n levels is $\frac{1}{2^n}$ and this tends to 0 very quickly.
The probability of $INSERT$ adding $2\log_2 n$ levels is $\frac{1}{2^{2\log_2 n}} = \frac{1}{n^2}$.
Consider the *Union Bound Theorem.*
Let E_j be the event where the j^{th} $INSERT$ puts its element in more than $2\log_2 n$ levels.
Then the probability that $E_1, \dots, E_j$ all occur at level one is at most $\sum_{i=1}^{j} \frac{1}{n^2} = \frac{1}{n}$.
Thus it is very unlikely for a skip list to have $2\log_2 n$ levels and the number of levels is $O(\log_2 n)$.

Remark 7.2 - *Constructing Skip List in Practice*
In practice you may want to consider setting a constant x so if a generated level has $< x$ elements you stop randomising.
This prevents having multiple duplicate layers higher up the skip list since duplicates become more likely with less elements and don't improve traversing at all.
Alternatively you could randomise the whole process and then evaluate the resulting skip list, removing duplicate layers.

2.8 Flow Networks

Definition 8.1 - *Flow Network*
A *Flow Network* is a directed graph where every edge has a non-negative *capacity.*
Exactly one node within a *Flow Network*is referred to as the *Source* & exactly one node as the *Sink.*
The aim of a *Flow Network* is to move the maximum amount of material from the source to the sink.

Definition 8.2 - *Capacity*
Capacity is a map from an edge to a non-negative real value.
When an edge has zero *Capacity* it is not shown.
The *Capacity* of an edge is the maximum amount of flow along that edge.
N.B. - $c : V \times V \to \mathbb{R}^{\geq 0}$.

Definition 8.3 - *Flow*
The *Flow* in the network G is a function $f : V \times V \to \mathbb{R}^{\geq 0}$.
The *Flow* function must satisfy the following conditions

1. $\forall\ u, v \in V\ 0 \leq f(u, v) \leq c(u, v)$.
   The flow along an edge is non-negative & cannot exceed the capacity of that edge;
2. And, $\forall\ u \in V \setminus \{s, t\}\ \sum_{v\in V} f(u, v) = \sum_{v\in V} f(v, u)$.
   The flow into all nodes, except the source & sink, must equal the flow out of the node.

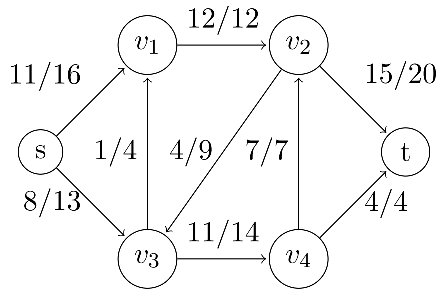
N.B. - The same network can have multiple flow functions.

Definition 8.4 - *Value*
The *Value* of a flow f is defined as

$$|f| := \sum_{v\in V} f(v, t) - f(t, v)$$

N.B. - This is the total flow into the terminal, less the total flow out of it.

Example 8.1 - *Flow Network*
Here is an example of a *Flow Network*.
The capacity & flow are denoted as f/c.

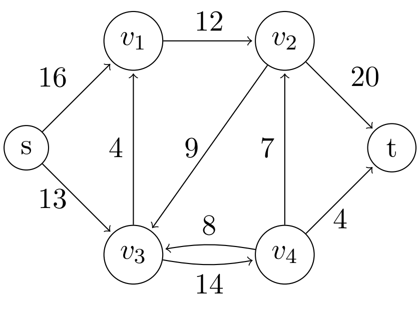


N.B. - Here $|f| = f(v_2, t) + f(v_4, t) - 0 = 15 + 4 = 19$.

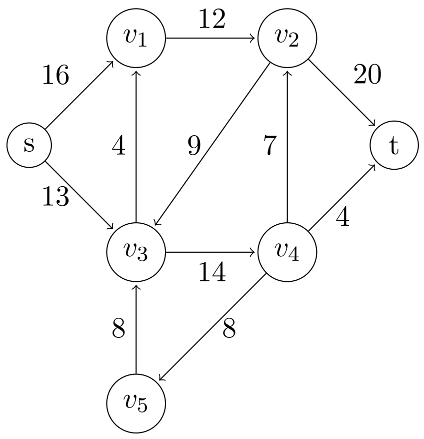
Definition 8.5 - *Antiparallel Edges*
An *Antiparrallel* edge is a pair of vertices u & v where $f(u, v) > 0$ & $f(v, u) > 0$.
N.B. - Each direction can have a different flow or capacity.

Proposition 8.1 - *Removing Antiparallel Edges*
We can remove *Antiparallel Edges* from a flow network by adding a new node and having one of the directions pass through that node, keeping the same capacity & flow.

Example 8.2 - *Removing Antiparallel Edges*
Here we remove the *Anti-Parallel* edge between v_3 & v_4.

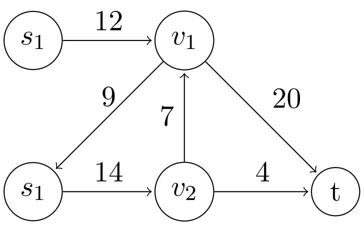


becomes

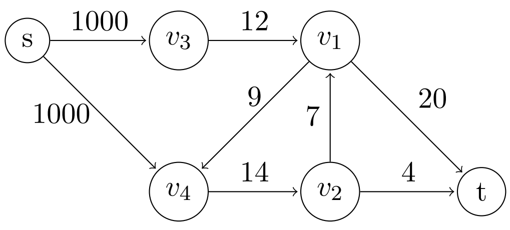


Proposition 8.2 - *Multiple Sources\Sinks*
We can transform a flow network with multiples sources or sinks to only have one.

Example 8.3 - *Multiple Sources\Sinks*



becomes



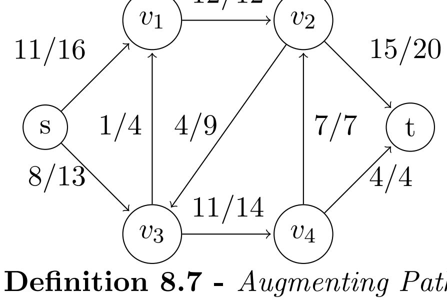
Definition 8.6 - *Residual Network*
Consider being given a flow network G & a flow f.
The *Residual Network* for this G & f is G_f where

$$E_f = \{(u,v) \in V \times V : c_f(u,b) \geq 0\}$$

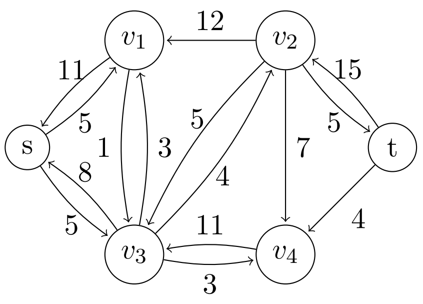
where

$$c_f(u,v) = \begin{cases} c(u,v) - f(u,v), & (u,v) \in E \\ f(v,u), & (v,u) \in E \\ 0, & otherwise \end{cases}$$

Example 8.4 - *Residual Network*
Flow Network



Residual Network



Definition 8.7 - *Augmenting Path*
An *Augmenting Path* is a simple path[1] from source to sink in the residual network.

Definition 8.8 - *Residual Capacity*
The *Residual Capacity* of an augmenting path p is given by $c_f(p) = min\{c_f(u,v) : (u,v) \in p\}$.

[1]See **Data Structures:Definition 1.2**

Definition 8.9 - *Augmentation of Flow*
Let f be a flow in a flow network G & f' be a flow in the residual network G_f.
The *Augmentation* of f by f' is a function $f \uparrow f' : V \times V \to \mathbb{R}$.
This function is defined such that $f \uparrow f'(u,v) = f(u,v) + f'(u,v) - f'(v,u)$.

Definition 8.10 - *Cut*
A *Cut* of a flow network is a partition of the vertex set into two sets, S & T, such that

$$s \in S,\ t \in T,\ S \cap T = \emptyset$$

Definition 8.11 - *Net Flow of a Cut*
Let f be a flow.
The *Net Flow* across a cut with sets S & T is defined as

$$f(S,T) := \sum_{u\in S}\sum_{v\in T} f(u,v) - \sum_{u\in S}\sum_{v\in T} f(v,u)$$

Definition 8.12 - *Capacity of a Cut*
The *Capacity* of a cut with sets S & T is defined as

$$c(S,T) := \sum_{u\in S}\sum_{v\in T} c(u,v)$$

Definition 8.13 - *Minimum Cut*
A *Minimum Cut* is a cut whose capacity is the minimum of all possible cuts a given flow network.
N.B. - There can be multiple such cuts.

Theorem 8.1 - *Net Flow of a Cut*
Let f be a flow with value $|f|$ in a flow network.
Let (S,T) be any cut of the same flow network.
Then $f(S,T) = |f|$.

Proof 8.1 - *Theorem 8.1*
This is a mathematical proof for ***Theorem 8.1****.*

$$\begin{array}{rcl}
|f| & = & \sum_{v\in V} f(s,v) - \sum_{v\in V} f(v,s) \\
 & = & \sum_{v\in V} f(s,v) - \sum_{v\in V} f(v,s) \\
 & + & \sum_{u\in S\setminus\{s\}} \sum_{v\in V} [f(u,v) - f(v,u)] \\
 & = & \sum_{v\in V} [f(s,v) + \sum_{u\in S\setminus\{s\}} f(u,v)] \\
 & - & \sum_{v\in V} [f(v,s) + \sum_{u\in S\setminus\{s\}} f(v,u)] \\
 & = & \sum_{v\in V} [\sum_{u\in S} f(u,v) - \sum_{u\in S} f(v,u)] \\
 & = & \sum_{v\in V} [\sum_{u\in S} f(u,v) - \sum_{u\in S} f(v,u)] \\
 & + & \sum_{v\in T} [\sum_{u\in S} f(u,v) - \sum_{u\in S} f(v,u)] \\
 & = & \sum_{v\in S}\sum_{u\in S} f(u,v) - \sum_{v\in S}\sum_{u\in S} f(v,u) \\
 & + & \sum_{v\in T}\sum_{u\in S} f(u,v) - \sum_{v\in T}\sum_{u\in S} f(v,u) \\
 & = & \sum_{v\in T}\sum_{u\in S} f(u,v) - \sum_{v\in T}\sum_{u\in S} f(v,u) \\
 & = & \sum_{u\in S}\sum_{v\in T} f(u,v) - \sum_{u\in S}\sum_{v\in T} f(v,u) \\
 & = & f(S,T)
\end{array}$$

N.B. - This is non-examinable.

Theorem 8.2 - *Max-Flow Min-Cut-Theorem*
The maximum flow value of a flow network is equal to its the minimum cut capacity.

Proof 8.2 - *Theorem 8.2*
Suppose f is a maximum flow in a flow network G.
By **Theorem 8.1**, $|f| = f(S,T)\ \forall$ cuts $(S,T) \in G$.
Then $|f| \leq c(S,T)\ \forall$ cuts $(S,T) \in G$.
G_f cannot have any paths from $s \to t$, as if there was one you could augment f along the path to create a flow with greater value.
Let S be the set of vertices u for which there is a path $s \to u$ in G_f.
Let $T = V - S$.
Suppose $u \in S,\ v \in T$.
Then (u,v) cannot be in G_f, otherwise v would be in S.
If $(u,v) \in G$ then $f(u,v) = c(u,v)$.
If $(v,u) \in G$ then $f(v,u) = 0$.

$$\begin{array}{rcl} |f| = f(S,T) & = & \sum_{u\in S}\sum_{v\in T} f(u,v) - \sum_{u\in S}\sum_{v\in T} f(v,u) \\ & = & \sum_{u\in S}\sum_{v\in T} c(u,v) \\ & = & c(S,T) \end{array}$$

We know that $|f| \leq c(S',T')\ \forall$ cuts $(S',T') \in G$.
Thus $c(S,T)$ must be the minimum cut capacity.

Remark 8.1 - *Modelling with Flow Networks*
Flow Networks are used to model road networks, computer networks, sewers etc.

Proposition 8.3 - *Max Flow-Min Cut Theorem for Image Segmentation*
The *Max Flow-Min Cut Theorem* can be used for image segmentation.
The aim of image segmentation is to split an image into its foreground & background.
Represent the pixels as nodes, creating edges to neighbouring pixels and setting the weight of each edge to be the difference in colour between the pixels.
Adding two new nodes, one for source & one for sink, which connect to all pixel nodes we can now perform *Max Flow-Min Cut* to create a cut.
N.B. - This is non-examinable.

2.9 Miscellaneous

Definition 9.1 - *Finite State Machine*
A *Finite State Machine* is a computation model used to simulate sequential logic.
*Finite State Machine*s must have the follow features

i) A finite set of states Q;

ii) An initial state $q_0 \in Q$;

iii) A set of accepting states $A \subset Q$;

iv) An input alphabet Σ; And,

v) A transition function $\delta : Q \times \Sigma \to Q$.

Remark 9.1 - *Applications of Finite State Machines*
Finite state machines can be used to detect regular expression patterns[1].
N.B. - A coded example of this can be seen in `programs/FiniteStateMachine.java` and `programs/PatternMatcher.java:finiteStateMachine`.

[1]See **Algorithms:Proposition 4.2**

Definition 9.2 - *Prefix Table*
The *Prefix Table*, π, for a string S is the integer array where $\pi[i]$ gives the length of the longest substring of $S[0,\dots,i-1]$ which is both a suffix and a prefix to $S[0,\dots,i]$.
Note that this substring cannot be equal to $S[0,\dots,i]$.

Proposition 9.1 - *Pseudocode for Constructing a Prefix Table*
A *Prefix Table*, for the string S, can be constructed using the following pseudocode[1]

```
PREFIX(S)
π=[]
for (i=1; i<S.length; i++):
  c=0
  π[i]=0
  for (j=1; j<i; j++:
    prefix=S[0,j]
    suffix=S[i-j,i];
    if (prefix==suffix):
      π[i]=c
return π
```

N.B. - This algorithm assumes strings are zero-indexed.

Example 9.1 - *Constructing Prefix Table*
Here we shall construct a prefix table for the string *"aaabbb"*.

Substring - *"a"*
Set $\pi[0]=0$.
Prefix cannot be whole string. Stop.

Substring - *"aa"*
Set $\pi[1]=0$.
prefix= *"a"*== *"a"*=suffix. Continue.
Set $\pi[1]=1$.
Prefix cannot be the whole string. Stop

Substring - *"aaa"*
Set $\pi[2]=0$.
prefix= *"a"*== *"a"*=suffix. Continue.
Set $\pi[2]=1$.
prefix= *"aa"*== *"aa"*=suffix. Continue.
Set $\pi[2]=1$.
Prefix cannot be the whole string. Stop.

Substring - *"aaab"*
Set $\pi[3]=0$.
prefix= *"a"*$\neq$ *"b"*=suffix. Stop.

Substring - *"aaabb"*
Set $\pi[4]=0$.
prefix= *"a"*$\neq$ *"b"*=suffix. Stop.

[1]See `programs\PrefixTable.java`

Substring - *"aaabbb"*
Set $\pi[5] = 0$.
prefix= *"a"*$\neq$ *"b"*=suffix. Stop.

The Prefix Table for *"aaabbb"* is $\pi = [0, 1, 2, 0, 0, 0]$.

Definition 9.3 - *Bad Match Table*
A *Bad Match Table* for a string is a map from each character in the string's alphabet to an integer.
This integer is the number of places you should shift the pattern along if the associated character is the character from the target text which causes a mismatch.

Proposition 9.2 - *Pseudocode for Constructing a Bad Match Table*
A *Bad Match Table* for the string S can be constructed using the following pseudocode[1]

```
BadMatchTable(S)
table=[]
table['*']=S.length
for (i=0; i<S.length; i++):
  char=S[i]
  table[char]=i
return table
```

Example 9.2 - *Constructing Bad Match Table*
Here we shall construct a bad match table for the string *"tooth"*.

Set $table['*'] = 5$.
Set $table['t'] = 0$.
Set $table['o'] = 1$.
Set $table['o'] = 2$.
Set $table['t'] = 3$.
Set $table['h'] = 4$.

The Bad Match Table for *"tooth"* is $['t' : 3,\ 'o' : 2,\ 'h' : 4,\ '*' : 5]$.

[1]See `programs\BadMatchTable.java`

3 Algorithms

3.1 Sorting

Definition 1.1 - *Insertion Sort*
Insertion Sort is an in-place sorting algorithm.
Insertion Sort can be performed on the ordable array A with the following pseudocode.[1]

```
INSERTION-SORT(A)
for (j=1; j<A.length; j++):
   key=A[j]
   i=j-1
   while (i>-1 && A[i]>key):
      A[i+1]=A[i]
      i--
   A[i+1] = key
return A
```

Theorem 1.1 - *Asymptotic Analysis of Insertion Sort*
The average case run-time complexity for insertion sort is $\in O(n^2)$.
The best case run-time complexity for insertion sort is $\in O(n)$.
The space complexity for insertion sort is $\in O(n)$.

Example 1.1 - *Insertion Sort*
Here we sort the list $[3, 4, 5, 1, 2]$.
3 is already sorted.
Sort 4.
Compare 4 & 3. $4 > 3$ Stop.
$[3, 4, 5, 1, 2]$.
Sort 5.
Compare 5 & 4. $5 > 4$ Stop.
$[3, 4, 5, 1, 2]$.
Sort 1.
Compare 1 & 5. $1 < 5$ Swap.
Compare 1 & 4. $1 < 4$ Swap.
Compare 1 & 3. $1 < 3$ Swap.
End of list. Stop.
$[1, 3, 4, 5, 2]$.
Sort 2.
Compare 2 & 5. $2 < 5$ Swap.
Compare 2 & 4. $2 < 4$ Swap.
Compare 2 & 3. $2 < 3$ Swap.
Compare 2 & 1. $2 > 1$ Stop.
$[1, 2, 3, 4, 5]$.

Proof 1.1 - *Correctness of Insertion Sort*
Here we prove that *Insertion Sort* is a loop invariant.
Initialisation
Set $j = 2$ then the array $A]1, \dots, j-1]$ has only one element, thus it is sorted.
Maintenance
The *for* loop moves $A[j-1], A[j-2], \dots$ to the right until it finds the position for $A[j]$ so $A[1, \dots, j]$ is sorted. Thus invariance holds.

[1]See `programs\Sort.java`

Termination
Since the loop has a predetermined termination value for j then termination is quaranteed.
Thus invariance holds.
Since we have proved *initialisation*, *maintenance* & *termination* we have proved that insertion sort is a preserved invariant.

Definition 1.2 - *Merge Sort*
Merge Sort is an in-place sorting algorithm.
Merge Sort can be performed on the ordable array A with the following pseudocode.[1]

```
MERGE-SORT(A,l,r)
n=A.length
if (n==1):
  return A
MERGE-SORT(A,l,n/2)
MERGE-SORT(A,n/2,r)
return MERGE(A,l,n/2,r)

MERGE(A,l,m,r)
L=A[l, m]
R=A[m, r]
i=0; j=0; k=0
while (i<m-l+1 && j<r-m):
  k++
  if (L[i]<=R[j]):
    A[k]=L[i]
    i++
  else:
        A[k]=R[j]
        j++
while (i<m-l+1):
  A[k]=L[i]
  i++; k++
while (j<r-m):
  A[k]=R[j]
  j++; k++
return A
```

N.B. - l is the left bound & r is the right bound. Initially these are set so $l = 0$ & $r = A.length$.

Theorem 1.2 - *Asymptotic Analysis of Merge Sort*
The run-time complexity for merge sort is $\in O(n \log_2 n)$.
The space complexity for insertion sort is $\in O(n)$.

Example 1.2 - *Merge Sort*
Here we sort the list $[3, 4, 5, 1, 2]$.
Split $[3, 4, 5, 1, 2]$ into $[3, 4, 5]$ & $[1, 2]$.
Split $[3, 4, 5]$ into $[3, 4]$ & $[5]$.
Split $[3, 4]$ into $[3]$ & $[4]$.
Merge $[3]$ & $[4]$ into $[3, 4]$.
Merge $[3, 4]$ & $[5]$ into $[3, 4, 5]$.

[1]See `programs\Sort.java`

Split $[1, 2]$ into $[1]$ & $[2]$.
Merge $[1]$ & $[2]$ into $[1, 2]$.
Merge $[3, 4, 5]$ & $[1, 2]$ into $[1, 2, 3, 4, 5]$.

Proof 1.2 - *Run-Time Complexity for Merge Sort*
This is a proof by strong induction.
Base Case
$n = 1$ $a = T(1)$
Inductive Hypothesis
The time complexity of merge sort for a list of n objects is given by

$$T(m) = am + bm\log_2 m \ \forall m < n,\ \ a, b \in \mathbb{R}$$

Inductive Step

$$\begin{array}{rcl} T(n) & = & 2T\left(\frac{n}{2}\right) + cn \\ \text{By hypothesis} & = & 2\left(a\frac{n}{2} + b.\frac{n}{2}.\log_2\left(\frac{n}{2}\right)\right) + cn \\ & = & an + bn\log_2 n - bn + cn \\ \text{For } b = c & = & an + bn.\log_2 n \end{array}$$

This is true $\forall$ n with $a = T(1)$ & $b = c$.

By the theorem of strong induction, $T(n)$ is the upper bounded for the worst case time for merge sort.

$$T(n) < (T(1) + c)n\log_2(n) \implies T(n) \in O(n\log_2(n))$$

Definition 1.3 - *Heap Sort*
Heap Sort is an out-of-place sorting algorithm.
Heap Sort can be performed on the ordable array A with the following pseudocode.

```
HEAP-SORT(A)
p=PRIORITY-QUEUE
B=[]
for i=0 to A.length:
  p.INSERT(A[i])
for i=0 to A.length:
  B[i]=p.EXTRACT-MIN()
return B
```

Example 1.3 - *Heap Sort*
Here we sort the list $[3, 4, 5, 1, 2]$ using *Heap Sort.*
We shall implement the priority queue with a binary heap.

Insert 3.

```
3
```

Insert 4.

```
3
|
4
```

Insert 5.

```
   3
  / \
 4   5
```

Insert 1.
```
  1
 / \
3   5
|
4
```
Insert 2.
```
    1
   / \
  2   5
 / \
4   3
```
Extract-min() = 1. $[1]$.
```
  2
 / \
3   5
|
4
```
Extract-min() = 2. $[1,2]$.
```
  3
 / \
4   5
```
Extract-min() = 3. $[1,2,3,]$.
```
4
|
5
```
Extract-min() = 4. $[1,2,3,4]$.
```
5
```
Extract-min() = 5. $[1,2,3,4,5]$.
The priority queue is now empty so heap sort is complete.
$[1,2,3,4,5]$.

Theorem 1.3 - *Asymptotic Analysis on Heap Sort*
When *Heap Sort* is performed using a binary heap, with the minimum-heap property, its run-time complexity is $\in O(n\log_2 n)$.
The space complexity for heap sort is $\in O(n)$.

3.2 Stable Matching

Definition 2.1 - *Stable Match*
A matching is considered a *stable match* if, either:

- Both members prefer their partner over all others. Or,
- If one/both members prefer another partner, then this preferred partner prefers their current partner over the new suitor.

N.B. - These situations are the same.

Definition 2.2 - *Stable Matching Problem*
Consider having two sets of people, X & Y.
Each person has a preference order for whom they wish to pair up with in the other group. and you are matching one member of X with one member of Y.
The *Stable Matching Problem* is to find a group of pairings where every pair is a *stable match.*

Definition 2.3 - *Feasible Pairing*
A pair (x, y) is a *Feasible Pairing* if there exists a stable matching of the sets X & Y which includes (x, y).

Theorem 2.1 - *Gale-Shapely Algorithm*
The *Gale-Shapely Algorithm* produces a set of pairings from two sets, such that all the pairings are stable matchings.
The *Gale-Shapely Algorithm* can be performed on the groups X & Y using the following process.[1]

1) All members of X & Y produce a preference order for the members of the other set.
2) All members of X propose to their first preference.
3) Make all members of Y now evaluates their proposals:
    i) If they have only one proposal, they accept it.
    ii) If they have multiple proposals, they accept the partner they find most preferable.
4) All those in X who don't have a partner propose to their next most preferable partner.
5) Everyone in Y, including those who currently have a partner, evaluate their new proposals in the same way as *3)*.
6) Repeat *4)* & *5)* until all members have a partner.

N.B. - If a member of Y receives a more preferable proposal, they drop their current partner for the most preferable new proposal.

Example 2.1 - *Gale-Shapely Algorithm*
Consider an example of matching 4 men & 4 women for a dance, where men propose to the women & the women evaluate these offers.
Initial Preferrence table

Frank	Kate	Mary	Rhea	Jill
Dennis	Mary	Jill	Rhea	Kate
Mac	Kate	Rhea	Jill	Mary
Charlie	Rhea	Mary	Kate	Jill

Rhea	Frank	Mac	Dennis	Charlie
Mary	Mac	Charlie	Dennis	Frank
Kate	Dennis	Mac	Charlie	Frank
Jill	Charlie	Dennis	Frank	Mac

In the first pass:

1. Frank propses to Kate.
2. Dennis proposes to Mary.
3. Mac Proposes to Kate.
4. Charlie proposes to Rhea.

[1]See `programs\GaleShapely.java`

Since Mary & Kate only recieved one proposal each, they accepted these proposals.
Kate has recieved two offers, since she prefers Mac over Frank she accepts his offer.

Frank	Kate	Mary	Rhea	Jill
Dennis	Mary	Jill	Rhea	Kate
Mac	Kate	Rhea	Jill	Mary
Charlie	Rhea	Mary	Kate	Jill

Rhea	Frank	Mac	Dennis	Charlie
Mary	Mac	Charlie	Dennis	Frank
Kate	Dennis	Mac	Charlie	Frank
Jill	Charlie	Dennis	Frank	Mac

In the second pass Frank is the only man without a partner, so he proposes to his next most preferrable which is Mary.
Mary prefers Dennis, her current partner, to Frank so does not change.

Frank	Kate	Mary	Rhea	Jill
Dennis	Mary	Jill	Rhea	Kate
Mac	Kate	Rhea	Jill	Mary
Charlie	Rhea	Mary	Kate	Jill

Rhea	Frank	Mac	Dennis	Charlie
Mary	Mac	Charlie	Dennis	Frank
Kate	Dennis	Mac	Charlie	Frank
Jill	Charlie	Dennis	Frank	Mac

Frank is still the only man without a partner, so proposes to Rhea.
Rhea prefers Frank over Charlie so drops Charlie to match with Frank.

Frank	Kate	Mary	Rhea	Jill
Dennis	Mary	Jill	Rhea	Kate
Mac	Kate	Rhea	Jill	Mary
Charlie	Rhea	Mary	Kate	Jill

Rhea	Frank	Mac	Dennis	Charlie
Mary	Mac	Charlie	Dennis	Frank
Kate	Dennis	Mac	Charlie	Frank
Jill	Charlie	Dennis	Frank	Mac

Charlie is now the only man without a partner, so proposes to Mary. Mary prefers Charlie over Dennis, so swaps.

Frank	Kate	Mary	Rhea	Jill
Dennis	Mary	Jill	Rhea	Kate
Mac	Kate	Rhea	Jill	Mary
Charlie	Rhea	Mary	Kate	Jill

Rhea	Frank	Mac	Dennis	Charlie
Mary	Mac	Charlie	Dennis	Frank
Kate	Dennis	Mac	Charlie	Frank
Jill	Charlie	Dennis	Frank	Mac

Dennis is now the only man without a parner, so proposes to Jill.
Jill is currently without a partner so accepts.

Frank	Kate	Mary	Rhea	Jill
Dennis	Mary	Jill	Rhea	Kate
Mac	Kate	Rhea	Jill	Mary
Charlie	Rhea	Mary	Kate	Jill

Rhea	Frank	Mac	Dennis	Charlie
Mary	Mac	Charlie	Dennis	Frank
Kate	Dennis	Mac	Charlie	Frank
Jill	Charlie	Dennis	Frank	Mac

Everyone now has a partner so the algorithm terminates.

Remark 2.1 - *Proposal Order effect on Gale-Shapely Algorithm*
The result of the *Gale-Shapely Algorithm* does not depend on the order that the proposers propose in.

Proposition 2.1 - *Everyone has a Partner at Termination*
Once the *Gale-Shapely Algorithm* terminates every member of both sets has a partner.

Proof 2.1 - *Everyone has a Partner at Termination*
This is a proof by contradiction.
Let X & Y be sets where X proposes to Y.
By the preconditions of the algorithm X & Y have the same number of members.
Suppose the *Gale-Shapely Algorithm* has terminated and not everyone has a partner.
Then, there must be at least one unmatched member from each set.
The unmatched members of X must have made a proposal to every member of Y, otherwise the algorithm would not have terminate.
Thus the unmatched members of Y have received at least one proposal, which they have to accept.
Thus they are matched.
This is a contradiction.

Proposition 2.2 - *All Pairings are Stable Matches*
Once the *Gale-Shapely Algorithm* terminates every pairing in the result is a stable matching.

Proof 2.2 - *All Pairings are Stable Matches*
This is a proof by contradiction.
Let X & Y be sets where X proposes to Y.
Let $x_0, x_1 \in X$ & $y_0, y_1 \in Y1$ where x_0 prefers y_1 over y_0 and y_1 pefers x_0 over x_1.
Suppose the algorithm has terminated and produced the unstable paring (x_0, y_0).
Since x_0 prefers y_1 to y_0, they would propose to y_1 before y_0.
Then, since y_1 prefers x_0 to x_1 they would accept this offer and would never drop x_0 to be with x_1.
Thus this final state is impossible given these preferences & this is a contradiction.

Proposition 2.3 - *Rejected Partners can never form Feasible Pairs*
Let X & Y be sets and $x \in X$ & $y \in Y$.
When the *Gale-Shapely Algorithm* is applied to X & Y if x rejects an offer from y at any point during then the pairing, (x, y), is not feasible.

Proof 2.3 - *Rejected Partners can never form Feasible Pairings*
This is a proof by strong induction.
Let X & Y be sets where X proposes to Y.
Let $x_0, x_1 \in X$ & $y_0, y_1 \in Y$.
Base Case
The first offer does not result in a rejection.
Inductive Assumption
If a proposal is rejected before offer $\#n$, it was for a non-feasible pair.
Inductive Case
Suppose, in response to the $\#n$ offer, y_0 rejects x_0.
Then y_0 must have recieved a more preferable offer from x_1, as one of offers $\#1, \dots, \#n-1$.
Consider any matching M containing (x_0, y_0).
Let (x_1, y_1) be a pairing in M.
Then we have two cases:

Case 1 - x_1 prefers y_0 to y_1.
Since y_0 prefers x_1 to x_0 then (x_0, y_1) is an unstable pair for M.
Then M would be unstable.

Case 2 - x_1 prefers y_1 to y_0.
x_1 must have made an offer to y_1, before it made an offer to y_0, and y_1 must have rejected this offer.
By the strong inductive hypothesis (x_1, y_1) is not feasible.
Then M would be unstable.

Proposition 2.4 - *Gale-Shapely Algorithm Favours Proposers*
The *Gale-Shapely Algorithm* matches each member of the proposing set to their favourite member of the other set with whom they have a feasible match.

Proof 2.4 - *Gale-Shapely Algorithm Favours Proposers*
Let X & Y be sets where X proposes to Y.
Let $x \in X$ & $y \in Y$.
The algorithm pairs x with a partner, y, for which (x, y) is feasible.
Let $best(x)$ be the first feasible partner to which x makes an offer.
By **Proposition 2.3** $best(x)$ does not reject x.

Proposition 2.5 - *Gale-Shapely Algorithm Disfavours Recipients*
The *Gale-Shapely Algorithm* matches each member of the receiving set to their least preferable member of the other set with whom they have a feasible match.

Proof 2.5 - *Gale-Shapely Algorithm Disfavours Recipients*
Let X & Y be sets where X proposes to Y.
Let $x_0, x_1 \in X$ & $y \in Y$.
By **Proposition 2.4**, the *Gale-Shapely Algorithm* assigns y to x_0 for which $y = best(x)$.
Consider any stable matching M.
Let x_1 be paired with y in M.
Then y prefers x_0 to its partner.
y cannot prefer x_0 to x_1, as this would make (x_0, y) an unstable pair.
Thus $y = worst(x)$.

Remark 2.2 - *Implementation of Gale-Shapely Algorithm - Data Structures*
For the *Gale-Shapely Algorithm* it is useful to take preference orders in as maps from a persons to name to a in-order list of their preferences.
For the proposing group it helps to take their preference order in as a queue or stack.

Definition 2.4 - *Inverse Preference List*
An *Inverse Preference list* is a list where the i^{th} element of the list is the preference order for the i^{th} option.
N.B. - When implementing the *Gale-Shapely Algorithm* it is efficient to have an inverse preference list.

Example 2.2 - *Inverse Preference List*

```
list    =[2,3,5,1,4]
inverse=[4,1,2,5,3]
```

Remark 2.3 - *Applications of Gale-Shapely Algorithm*
The Gale-Shapely algorithm has been applied:

1. Matching US students to schools, reducing preferred matches by 90%.
2. By Akami to match web stream with servers.
3. Matching organ donors & recipients in the NHS.

3.3 Order Statistics

Definition 3.1 - *Order Statistics Problem*
Consider having an unordered list of ordable elements & an integer i.
The *Order Statistics Problem* is to find the i^{th} smallest value of the list.

Proposition 3.1 - *Naïve Approach*
The *Naïve Approach* to solve the *Order Statistics Problem* may be to sort the list and then select the i^{th} value in the list.
However the run-time complexity for the best sorting algorithms are $\Theta(n \log_2 n)$.
N.B. - The *Order Statistics Problem* can be solved with a run-time complexity $\in O(n)$, using a specific algorithms.

Definition 3.2 - *Partition*
A *Partition* is made in a list of ordable elements by taking a pivot and moving all elements less than the pivot to the left of the pivot, and all those greater than it to the right.
Algorithms that *Partition* a list generally choose the pivot randomly and return the final index of the pivot.
A *Partition* on the elements between the indexes l & r of the list A can be performed with the following pseudocode.[1]

```
PARTITION(A, l , r)
p=RANDOM( l , r)
x=A[p]
swap A[p] and A[r-1]
i=l
for (j=l; j<r; j++):
   if (A[j]≤x):
      swap(arr, i, j)
      i++
return i-1
```

N.B. - This is used in quick sort.

Example 3.1 - *Partition*
Here we shall perform PARTITION($[5, 4, 6, 3, 1, 2, 8, 7]$,0,7).
Lets randomly choose 6 for the pivot.
Set $i = 0$.
Swap 6 & 7.
$[5, 4, 7, 3, 1, 2, 8, 6]$.
Compare 5 & 6.
$5 \leq 6$. Swap 5 and index $i = 0(5)$.
$[5, 4, 7, 3, 1, 2, 8, 6], i = 1$.
Compare 4 & 6.
$4 \leq 6$. Swap 4 and index $i = 1(4)$.
$[5, 4, 7, 3, 1, 2, 8, 6], i = 2$.
Compare 7 & 6.
$7 \not\leq 6$. No change.
$[5, 4, 7, 3, 1, 2, 8, 6], i = 2$.
Compare 3 & 6.
$3 \leq 6$. Swap 3 and index $i = 2(7)$.
$[5, 4, 3, 7, 1, 2, 8, 6], i = 3$.
Compare 1 & 6.
$1 \leq 6$. Swap 1 and index $i = 3(7)$.
$[5, 4, 3, 1, 7, 2, 8, 6], i = 4$.

[1]See `programs\OrderStatistics.java`

Compare 2 & 6.
$2 \leq 6$. Swap 2 and index $i = 4(7)$.
$[5, 4, 3, 1, 2, 7, 8, 6], i = 5$.
Compare 8 & 6.
$1 \not\leq 6$. No Change.
$[5, 4, 3, 1, 2, 7, 8, 6], i = 5$.
Compare 6 & 6.
$6 \leq 6$. Swap 6 and index $i = 5(7)$.
$[5, 4, 3, 1, 2, 6, 8, 7], i = 6$.
Done, return 5.

Definition 3.3 - *Random Select Algorithm*
The *Random Select Algorithm* can be used to solve the *Order Statistics*.
The *Random Select Algorithm* is performed by using *PARTITION* repeatedly, decreasing the range of the bounds each time.
The *Random Select Algorithm* can be performed to find the i^{th} smallest value in the list A with the following pseudocode.[1]

```
RANDOM SELECT(A,l,r,i)
 if l==r
   return A[l]
p=PARTITION(A,l,r)
 if (i==p):
   return A[i]
 if (p>i):
   return RANDOM-SELECT(A,l,p-1,i)
 return RANDOM-SELECT(A,p+1,r,i)
```

N.B. - Initially $l = 0$ and $r = A.length - 1$.

Example 3.2 - *Random Select*
Here we shall perform RANDOM-SELECT($[5, 4, 2, 1, 3]$,0,4,2).
$0 \neq 4$.
PARTITION($[5, 4, 2, 1, 3]$,0,4).
Lets randomly choose 4 for the pivot.
This returns $[2, 1, 3, 4, 5]$ and $p = 3$.
$p = 3 > 2 = i$

RANDOM-SELECT($[2, 1, 3, 4, 5]$,0,2,2) $0 \neq 2$.
PARTITION($[2, 1, 3, 4, 5]$,0,2).
Lets randomly choose 1 for the pivot.
This returns $[1, 2, 3, 4, 5]$ and $p = 0$.
$p = 0 < 2 = i$.

RANDOM-SELECT($[1, 2, 3, 4, 5]$,1,2,2)
$1 \neq 2$.
PARTITION($[1, 2, 3, 4, 5]$,1,2)
Lets randomly choose 3 for the pivot.
This returns $[1, 2, 3, 4, 5]$ and $p = 2$.
$p = 2 \equiv 2 = i$.
Done, return 3.

[1] See `programs\OrderStatistics.java`

Proposition 3.2 - *Worst Case Run-Time Complexity of RANDOM-SELECT*
Here we assume that all the elements of A are different.
Suppose that after performing *PARTITION* there is only one element on one side of the pivot and the rest of the elements are on the other side.
The *worst case scenario* is when we now need to perform $RANDOM-SELECT$ on the larger side.
This gives a time complexity of

$$T(n) = T(n-1) + f(n) \quad f(n) \in \Theta(n)$$

Thus $T(n) \in \Theta(n^2)$.

Proposition 3.3 - *Best Case Run-Time Complexity of RANDOM-SELECT*
Here we assume that all the elements of A are different.
The *best case scenario* occurs when exactly half the elements are on one side of the pivot and the other half is on the other side, after performing *PARTITION*.
This gives a time complexity of

$$T(n) = T\left(\frac{n}{2}\right) + f(n) \quad f(n) \in \Theta(n)$$

Analysing this with the *Master Theorem* we get $a = 1, b = 2, c = 1$ so $\log_b a = \log_2 1 = 0 < c$ so

$$T(n) \in \Theta(f(n)) \equiv T(n) \in \Theta(n)$$

Definition 3.4 - *Blum, Floyd, Pratt, Rivest, Tarjan Algorithm*
The *Blum, Floyd, Pratt, Rivest, Tarjan Algorithm* is a solution to the *Order Statistics Problem* than reduces the chance of incurring the *worst case scenario*.
The *BFPRT Algorithm* does this by choosing a better pivot.
The *BFPRT Algorithm* splits the list into groups of 5 elements, finds the medians of each group and then uses the median of the group medians as the pivot.
The *BFPRT Algorithm* can be preformed to find the i^{th} smallest value in the list A with the following pseudocode.[1]

```
BFPRT-SELECT(A,l,r,i)
Divide A[l,...,r] into groups of 5
Find median of each group
x=Median of group medians
Partition A[l,...,r] using x as pivot
k=index of x
if (i==k):
  return x
if (i<k):
  return BFPRT-SELECT(A,l,k,i)
return BFPRT-SELECT(A,k+1,r,i-k-1)
```

N.B. - The last group may have less than 5 elements.

[1] See `programs\OrderStatistics.java`

Example 3.3 - *Blum-Floyd-Pratt-Rivest-Tarjan Algorithm*
Here we shall perform BFPRT-SELECT([11,12,5,15,14,7,10,9,4,3,6,8,13],0,14,3).
Divide A[0,14] into [11,12,5,15,14], [7,10,9,4,3] & [6,8,13].
These have medians 12, 7 & 8.
The median of these is 8.
Partition A[0,14] around 8.
$[5, 7, 4, 3, 6, 8, 11, 12, 15, 14, 10, 9, 13]$, k=5.
i=3<5=k.

BFPRT-SELECT([5,7,4,3,6,8,11,12,15,14,10,9,13], 0, 5, 3).
Divide A[0,5]into [5,7,4,3,6].
This has median 5.
Partition A[0,5] around 5.
$[4, 3, 5, 6, 7]$, k=2.
i=3>2=k.

BFPRT-SELECT([4,3,5,6,7,8,11,12,15,14,10,9,13], 3, 5, 0).
Divide A[3,5] into [6,7].
This has median 6.
Partition A[3,5] around 6.
$[6, 7]$, k=0.
i=0≡0=k.
return 6.
Theorem 3.1 - *Average Case Run-Time Complexity for BFPRT-SELECT*
Breaking the elements into groups $\in \Theta(n)$.
Sorting one group $\in \Theta(1)$.
Finding all the medians $\in \Theta(n)$.
Finding median of medians $\in T(\lceil \frac{n}{r} \rceil)$.
Partitioning around pivot $\in \Theta(n)$.
So $T(\lceil \frac{n}{5} \rceil) + f(n) \leq T(\lceil \frac{n}{5} \rceil) + \alpha n,\ \alpha > 0$.

Theorem 3.2 - *Worst Case Run-Time Complexity of BFPRT-SELECT*
Here we assume that all elements of A are different.
At least half of the $\lceil \frac{n}{5} \rceil$ medians are less than equal to pivot x, only one is equal to x.
At most, one group has less than 5 elements.
So at least $3(\frac{1}{2}\lceil \frac{n}{5} \rceil - 2)$ elements are greater than x.
Thus the number of elements in the upper half is bounded by

$$n - 3\left(\frac{1}{2}\left\lceil \frac{n}{5} \right\rceil - 2\right) \leq \frac{7n}{10} + 6$$

And the lower half has at most $\frac{7n}{10} + 6$ elements.
By including the time to analyse these elements we find

$$T(n) \leq T\left(\left\lceil \frac{n}{5} \right\rceil\right) + T\left(\left\lfloor \frac{7n}{10} \right\rfloor + 6\right) + \alpha n \quad \alpha > 0$$

Proof 3.1 - *Worst Case Run-Time Complexity of BFPRT-SELECT - Strong Induction*
This is a proof by Strong Induction.
We have $T(n) \leq T\left(\lceil\frac{n}{5}\rceil\right) + T\left(\lfloor\frac{7n}{10}\rfloor + 6\right) + \alpha n$.
Here we will show that $\exists\ c \in \mathbb{R}^{>0}\ st\ T(n) \leq cn \implies T(n) \in O(n)$.
Base Case

$$n = 0 \implies T(0) = 0$$

Inductive Hypothesis

$$T(m) \leq cm\ \forall\ m \in \mathbb{N}_0^{\leq n}$$

Inductive Step

$$\begin{array}{rcll} T(n) & \leq & T(\lceil\frac{n}{5}\rceil) + T(\lfloor\frac{7n}{10}\rfloor + 6) + \alpha n & \\ \text{By hypothesis} & \leq & c(\lceil\frac{n}{5}\rceil) + c(\lfloor\frac{7n}{10}\rfloor + 6) + \alpha b & \text{if } n \geq 7 \\ & \leq & c(\frac{n}{5} + 1 + \frac{7n}{10} + 6) + \alpha n & \\ & \leq & cn & \text{if } c > max(20\alpha, 140) \end{array}$$

By picking $c > MAX(20\alpha, 140)$ st $T(m) \leq cm\ \forall\ m = [0, 6]$.
Result follows $\forall\ n \geq 0$.
So $T(n) \in O(n)$.

Proof 3.2 - *Worst Case Run-Time Complexity of BFPRT-SELECT - Akra-Bazzi Formula*
This proof uses the Akra-Bazzi Formula.
We have $T(n) \leq T\left(\lceil\frac{n}{5}\rceil\right) + T\left(\lfloor\frac{7n}{10}\rfloor + 6\right) + \alpha n$.
For $n > 120$ we have $\lfloor\frac{7n}{10}\rfloor + 6 \leq \frac{15n}{20} = \frac{3n}{4}$.
So $\forall\ n \geq 0,\ T(n) \leq U(n)$ where

$$U(n) = \begin{cases} T(n) & n \leq 120 \\ U(n) = U(\lceil\frac{n}{4}\rceil) + U(\lceil\frac{3n}{4}\rceil) + \alpha n & n > 120 \end{cases}$$

We can set the conditions for the *Akra-Bazzi Formula* to be

$$k = 2,\ a_1 = a_2 = 1,\ d_1 = \frac{1}{4},\ d_2 = \frac{3}{4},\ c = 1$$

These satisfy the conditions of the *Akra-Bazzi Formula.*
Set $(\frac{1}{5})^p + (\frac{3}{5})^p = 1 \implies p < 1$.
Thus $U(n) \in \Theta(n)$.
Since $T(n) \leq U(n)$ for large n we have $T(n) \in O(n)$.

3.4 Pattern Matching

3.4.1 Exact String Matching

Remark 4.1 - *Motivation*
String matching algorithms are used in web searching & DNA matching.

Definition 4.1 - *Exact String Matching Problem*
Consider having two strings a text, T; and, a pattern, P, where $T.length \geq P.length$.
The *Exact String Matching Problem* is to find the index of the first occurrence of the pattern in the text.

Remark 4.2 - *Evaluating String Matching Algorithms*
When evaluating string matching algorithms we take the number of comparisons as a measure of performance.
The fewer comparisons, the better performance.

Proposition 4.1 - *Naïve Approach*
The *Naïve Approach* may be to check the pattern against the text, character by character.
Then, whenever a mismatch is found shift the pattern along by *one* character and start evaluating again.
The *Naïve Approach* can be performed with pattern P & text T using the following pseudocode.[1]

```
NAIVE-MATCHER(P,T)
n = T.length
m = P.length
for (i=0; i+m<n; i++):
   if (P[1,...,m]==T[i+1,...,i+m]):
      return i+1
return null
```

Remark 4.3 - *Run-Time Complexity of N̈aive Method*
The *Naïve Method* matches up to the m^{th} character and restarts each time it gets a mismatch.
So the total number of comparisons is $(n-m+1)m$.
The worst case of this is when $m=\frac{n}{2}$ making the run-time complexity $\in\Theta(n^2)$.

Definition 4.2 - *Knuth-Morris-Pratt Algorithm*
The *Knuth-Morris-Pratt Algorithm* is a solution to the *Exact String Matching Problem.*
The *KMP Algorithm* uses a *Prefix Table*[2].
When a mismatch occurs, the *KMP Algorithm* uses data from the previous matches to skip past positions it knows will not match.
The *KMP Algorithm* can be performed on a text, T, and pattern, P, using the following pseudocode.[3]

```
KMP-MATCHER(P, T)
n=T.length
m=P.length
π=PREFIX(P)
q=-1
for (i=0; i<n; i++):
   while (q>-1 && P[q+1]≠T[i]):
      q=π[q]
   if (P[q+1]==T[i]):
      q++
   if (q==m):
      return(i-m+1)
return null
```

[1]See `programs\PatternMatcher.java:naive`
[2]See **Data Structures:Definition 9.2**
[3]See `programs\PatternMatcher.java:kmp`

Example 4.1 - *Knuth-Morris-Pratt Algorithm*
Here we shall perform $KMP - MATCHER(ccab, abccabb)$.
$n = 7,\ m = 4,\ \pi = [0, 1, 0, 0],\ q = 0$.
for loop iteration, $i = 0$
$q = -1 \equiv -1$ so while loop never runs.
$P[0] = c! = a = T[0]$.
$q = 0! = 4 = m$.
for loop iteration, $i = 1$
$q = -1 \equiv -1$ so while loop never runs.
$P[0] = c! = b = T[1]$.
$q = 0! = 4 = m$.
for loop iteration, $i = 2$
$q = -1 \equiv -1$ so while loop never runs.
$P[0] = c \equiv c = T[2]$.
$q = 1$.
$q = 1! = 4 = m$.
for loop iteration, $i = 3$
$q = 0 > -1$ but $P[1] = c \equiv c = T[3]$ so while loop never runs.
$P[1] = c \equiv c = T[3]$.
$q = 2$.
$q = 2! = 4 = m$.
for loop iteration, $i = 4$
$q = 1 > -1$ but $P[2] = a \equiv a = T[4]$ so while loop never runs.
$P[2] = a \equiv a = T[4]$.
$q = 3$.
$q = 3! = 4 = m$.
for loop iteration, $i = 5$
$q = 3 > -1$ but $P[4] = b \equiv b = T[5]$ so while loop never runs.
$P[4] = b \equiv b = T[5]$.
$q = 4$.
$q = 4 \equiv 4 = m$.
return $i - m = 5 - 4 + 1 = 2$.

Remark 4.4 - *Run-Time Complexity of Knuth-Morris-Pratt Algorithm*
When $q = 0$ there are at most n comparisons.
Since q only increases, by 1, after a successful comparison and only decrease if $q > 0$
then q is never negative.
So there are at most n unsuccessful comparison with $q > 0$.
Then there are no more than $2n$ comparisons.
Thus, the *Knuth-Morris-Pratt Algorithm* $\in O(n)$.
Moreover, *Knuth-Morris-Pratt Algorithm* $\in \Theta(n)$ since it performs at least n comparisons.

Remark 4.5 - *Applications of Knuth-Morris-Pratt Algorithm*
Since the *Knuth-Morris-Pratt Algorithm* never backtracks it can be used in strings which are streamed.

Definition 4.3 - *Boyer-Moore-Horspool Algorithm*
The *Boyer-Moore-Horspool Algorithm* is a solution to the *Exact String Matching Problem.*
The *BMH Algorithm* uses a *Bad Match Table.*[1]
The *BMH Algorithm* is designed to work better with large alphabets and patterns with little repetition.
The *BMH Algorithm* compares the pattern & string by starting with the last character of the pattern and then moving left.
The *BMH Algorithm* can be performed on a text T and a pattern P, using the following pseudocode.[2]

```
BMH(T,P)
 table=BadMatchTable(P)
 i=0; j=P.length-1
 while (i+j<T.length):
   if (P[j]==T[i+j]):
     if (j==0):
       return i
     j--
   else:
     step=j-table[T[i+j]]
     if (step<1):
       step=1
     i+=step
     j=P.length-1
```

Example 4.2 - *Boyer-Moore-Horspool Algorithm*
Here we shall performed $BMH(ccab, abccabb)$.
$table = ['c' : 1, 'a' : 2, 'b' : 3, '*' : 4]$.
$i = 0,\ j = 4 - 1 = 3$.
$i + j = 0 + 3 = 5 < 7 = T.length$ so while loop iterates.
$P[3] = b \neq c = T[3]$.
$step = 3 - table[c] = 3 - 1 = 2\ i = 0 + 2 = 2$.
$j = 4 - 1 = 3$.
$i + j = 2 + 3 = 5 < 7 = T.length$ so while loop iterates.
$P[3] = b \neq b = T[5]$.
$j = 3 \neq 0$.
$j = 2\ i + j = 2 + 2 = 4 < 7 = T.length$ so while loop iterates.
$P[2] = a \neq a = T[4]$.
$j = 2 \neq 0$.
$j = 1\ i + j = 2 + 1 = 3 < 7 = T.length$ so while loop iterates.
$P[1] = c \neq c = T[3]$.
$j = 1 \neq 0$.
$j = 0\ i + j = 2 + 0 = 2 < 7 = T.length$ so while loop iterates.
$P[0] = c \neq c = T[2]$.
$j = 0 \equiv 0$.
return i=2.

[1]See **Data Structures:Definition 9.3**
[2]See `programs\PatternMatcher.java:bmh`

Remark 4.6 - *Run-Time Complexity of Boyer-Moore-Horspool*
In the worst case scenario $m(n-m+1)$ comparisons are performed $\in O(n^2)$.
In the average case scenario $\in O(n)$.
In the best case scenario $\lfloor \frac{n}{m} \rfloor$ comparisons are made $\in O(n)$.

3.4.2 Regular Expression Pattern Matching

Proposition 4.2 - *Finite State Machines for Regular Expression Pattern Matching*
A *Finite States Machine*[1] can be constructed for parsing regular expressions[2].
Let P be a pattern which we want to construct a *Finite State Machine* to analyse for.
This can be achieved by defining the features of the *Finite State Machine* as[3]

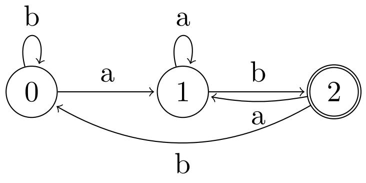
i) $Q = \{0, \dots, P.length\}$

ii) $q_0 = 0$

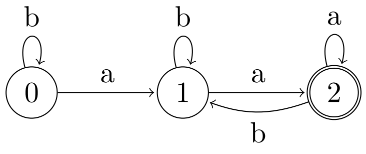
iii) $A = \{p.length\}$

iv) For $q \in Q,\ c \in \Sigma$ let $\delta(q,c) =$ longest suffix of "$P[1 \dots q]c$" which is also a prefix of P.

Example 4.3 - *Finite State Machine for Regex Pattern Matching*
Below is an automata representation of a finite state machine matching for the pattern *ab*.



Below is an automata representation of a finite state machine matching for the pattern $ab*a$.



Proposition 4.3 - *Pseudocode for Pattern Matching*
A *Finite State Machine* can be used to match a pattern P, against a text T, using the following pseudocode.[4]

```
FSM-MATCHER(T,P)
n = T.length
δ=FSM-BUILD(P)
s=0
for (i=0; i<T.length; i++):
   s=δ(s, T[i])
   if (s==δ.A):
      return(i - m)
```

[1]See **Data Structures:Definition 9.1**
[2]See **Reference:Definition 2.1**
[3]See `programs\FiniteStateMachine.java`
[4]See `programs\PatternMatcher.java:finiteStateMachine`

3.5 Fourier Transformations

Definition 5.1 - *Fourier Transformations*
Fourier Transformations are used to multiply two large polynomials or integers together.

Theorem 5.1 - *Horner's Method*
Horner's Method is a technique for evaluating polynomials.
Given a polynomial $f(x) = \sum_{i=1}^{n} a_i x^i$ and a value x_0 to evaluate at.
Horner's Method states that you can express $f(x_0)$ as

$$f(x_0) = a_0 + x(a_1 + x(a_2 + \cdots + x(a_{n-1} + a_n x)))$$

Horner's Method can be performed on polynomial A of degree n in coefficient representation at value x_0 by the following pseudocode.[1]

```
HORNER(A, n, $x_0$)
t=0
for (i=m-1; i$\geq$0; i--):
  t = (t*$x_0$)+A[i]
return t
```

Example 5.1 - *Horner's Method*
Consider the equation $5 + 4x + 3x^2 + 2x^3 + x^4$.
This can be written as $5 + x(4 + x(3 + x(2 + x)))$.

Remark 5.1 - *Run-Time Complexity of Horner's Method*
Horner's Method has a run-time complexity $\in O(n)$.

Proposition 5.1 - *Run-Time Complexity for Multiplying Polynomials*
The run-time complexity for multiplying coefficient represent polynomials $\in O(n^2)$.
The run-time complexity for multiplying two point-value represented polynomials $\in O(n)$.
The run-time complexity for converting between point-value & coefficient representations of polynomials $\in O(n)$.
Thus, the run-time complexity for converting to point-value representation then multiplying then converting back to coefficient representation $\in O(n)$.
Thus this process is preferable for large polynomials.

Proposition 5.2 - *Implementing Point-Value Multiplication*
Consider having two polynomials $A(x)$ & $B(x)$.
Let $n = MAX(A.degree,\ B.degree)$.
We can implement the process outlined in **Proposition 5.1** using the following process

1) Create coefficient representations of $A(x)$ and $B(x)$ with degree-bound $2n$ by padding high-orders with zeros.

2) Compute point-value representations of $A(x)$ and $B(x)$ of length $2n$.

3) Use point-value multiplication to compute a point-value representation of $C(x) = A(x).B(x)$

4) Create a coefficient representation of $C(x)$.

[1] See `programs\FourierTransform.java:horner`

Definition 5.2 - *Discrete Fourier Transform*
Discrete Fourier Transform converts a finite sequence of equally-spaced samples.
Let $A(x)$ be a polynomial of degree n.
To produce the *Discrete Fourier Transform* of $A(x)$ we evaluate $A(x)$ with the n roots of unity.
Define $y_k = A\left(\omega_n^k\right)$.
Then $y = (y_0, \dots, y_{n-1})$ is the *Discrete Fourier Transform* of $A(x)$.

Proposition 5.3 - *Pseudocode for Discrete Fourier Transform*
The *Discrete Fourier Transform* of a polynomial A, in coefficient form, of degree n can be produced by the following pseudocode.[1]

```
DFT(A,n)
y=[]
for (i=0; i<n; i++):
  $\omega$=rootOfUnity(n,i)
  y[i]=HORNER(A,n,$\omega$)
return y
```

Example 5.2 - *Discrete Fourier Transform*
Let $A(x) = 0 + 0x + x^2 - x^3$.
$A(x)$ is degree-bounded by 4.
The 4 roots of unity are $1, i, -1, -i$.
Evaluating at these we get

$$\begin{array}{rcccl} A(1) & = & 0+0+1-1 & = & 0 \\ A(i) & = & 0+0-1+i & = & i-1 \\ A(-1) & = & 0+0+1+1 & = & 2 \\ A(-i) & = & 0+0-1-i & = & -1-i \end{array}$$

So $y = (0, i-1, 2, -1-i)$ is the discrete fourier transform.

Definition 5.3 - *Fast Fourier Transform*
The *Fast Fourier Transform* is a faster way to calculate the *Discrete Fourier Transform* of a polynomial.

[1]See `programs\FourierTransform.java:dft`

Proposition 5.4 - *Pseudocode for Fast Fourier Transform*
The *Fast Fourier Transform* of a polynomial A, in coefficient form, of degree n can be produced by the following pseudocode. [1]

```
FFT(A,n)
if n == 1
  return A
else
  y=[]
  $\omega_n$ = $e^{2\pi i/n}$
  $\omega$ = 1
  $A^{[0]}$ = ($a_0, a_2, \dots, a_{n-2}$)
  $A^{[1]}$ = ($a_1, a_3, \dots, a_{n-1}$)
  $y^{[0]}$ = FFT($A^{[0]}, \frac{n}{2}$)
  $y^{[1]}$ = FFT($A^{[1]}, \frac{n}{2}$)
  for (k=0; k<$\frac{n}{2}$; k++):
    $y$[k]=$y^{[0]}$[k]+$\omega$*$y^{[1]}$[k]
    $y$[k+n/2]=$y^{[0]}$[k]−$\omega$*$y^{[1]}$[k]
    $\omega$ = $\omega$*$\omega_n$
  return y
```

Example 5.3 - *Fast-Fourier Transform*
Here we shall perform a *Fast Fourier Transform* on the polynomial $f(x) = x^2 - x^3$.
This shall be done by performing FFT([0,0,1,-1],4).

```
$n = 4 \neq 1$. Go to else clause.
$y = []$.
$\omega_n = e^{\frac{2\pi i}{4}} = i,\ \omega = 1$.
$A^{[0]} = [a_0, a_2] = [0, 1],\ A^{[1]} = [a_1, a_3] = [0, -1]$.
$y^{[0]} = FFT([0, 1], 2)$
      $2 \neq 1$. Go to else clause.
      y =[]
      $\omega_n = e^{\frac{2\pi i}{2}} = -1,\ \omega = 1$.
      $A^{[0]} = [0],\ A^{[1]} = [1]$
      $y^{[0]} = FFT([0], 1) = [0]$
      $y^{[1]} = FFT([1], 1) = [1]$
      k=0; k<1; k++
      y[0]=0+(1*1)=1
      y[1]=0−(1*1)=−1
    =[1,−1]
```

[1] See `programs\FourierTransform.java:fft`

```
y^{[1]} = FFT([0,-1],2)
        2 ≠ 1. Go to else clause.
        y =[]
        $\omega_n = e^{\frac{2\pi i}{2}} = -1,\ \omega = 1$.
        $A^{[0]} = [0],\ A^{[1]} = [1]$
        $y^{[0]} = FFT([0],1) = [0]$
        $y^{[1]} = FFT([-1],1) = [-1]$
        k=0; k<1; k++
        y[0]=0+(-1*1)=-1
        y[1]=0-(-1*1)=1
      =[-1,1]
k=0; k<2; k++
y[0]=1+(1*-1)=0
y[2]=1-(1*-1)=2
k++;
y[1]=-1+(i*1)=i-1
y[3]=-1-(i*1)=-1-i

y=[0,i,-1,2,-1,-i]
```

N.B. - This is the same result as **Example 5.2**.

Remark 5.2 - *Correctness of Fast Fourier Transform*
Consider the definitions of $A^{[0]}$ & $A^{[1]}$ in **Proposition 5.4**.
We have $A(x) = A^{[0]}(x^2) + xA^{[1]}(x^2)$.
This means we only need to evaluate at $(\omega_n^0)^2, (\omega_n^1)^2, \dots, (\omega_n^{n-1})^2$.
By the *Halving Rule*[1] for complex roots of unity we are now only evaluating at $\frac{n}{2}$ points rather than n.

Remark 5.3 - *Runtime Complexity of Fast Fourier Transform*
The recursive calls use half the input each & calculating the values of y_k and $y_{k+\frac{n}{2}} \in \Theta(n)$.
This gives a complexity equation of

$$T(n) = 2T\left(\frac{n}{2}\right) + \alpha n \in \Theta(n \log_2 n)$$

Remark 5.4 - *Maths from Pseudocode*
Here we consider the pseudocode in **Definition 5.3**.
Line 13 gives

$$\begin{array}{rcl} y_k & = & y_k^{[0]} + \omega_n^k y_k^{[1]} \\ & = & A^{[0]}(\omega_n^{2k}) + \omega_n^k A^{[1]}(\omega_n^{2k}) \\ & = & A(\omega_n^k) \end{array}$$

Line 14 gives

$$\begin{array}{rcl} y_{k+\frac{n}{2}} & = & y_k^{[0]} - \omega_n^k y_k^{[1]} \\ & = & y_k^{[0]} + \omega_n^{k+\frac{n}{2}} y_k^{[1]} \\ & = & A^{[0]}(\omega_n^{2k}) + \omega_n^{k+\frac{n}{2}} + A^{[1]}(\omega_n^{2k}) \\ & = & A^{[0]}(\omega_n^{2k+n}) + \omega_n^{k+\frac{n}{2}} + A^{[1]}(\omega_n^{2k+n}) \\ & = & A(\omega_n^{k+\frac{n}{2}}) \end{array}$$

[1]See **Reference:Theorem 3.2**

Definition 5.4 - *Inverse Fourier Transform*
The *Inverse Fourier Transform* takes a point-represented polynomial and produces a coefficient-representation.
This relies on the theorem that

$$a_i = \frac{1}{n}\sum_{j=0}^{n-1} y_j \omega_n^{-ij}$$

The process for performing a *Inverse Fourier Transform* is similar to computing the Discrete Fourier Transform, with the following amendments.

i) Switch the roles of a and y.

ii) Replace ω_N with ω_N^{-1}.

iii) Divide the final result by n.

N.B. - This has run-time complexity $\in \Theta(n \log_2 n)$.

3.6 Graphs

Proposition 6.1 - *Traversing Graphs*
When considering a general approach to *Traversing a Graph* we consider using a *bag.*
We can put a vertex into the bag, and we can take a vertex out of the *bag.*
Several different data structures can be used to implement the *bag*, each producing a different method for traversing.

Proposition 6.2 - *Pseudocode for General Traversal of Graphs*
The following pseudocode can be used visit every node of a graph G, starting at vertex s.
Here we don't know which what vertex the bag will give us.

```
TRAVERSE(G, s)
mark=[]
put $s$ into $bag$
while ($bag$ !empty):
   take $u$ from $bag$
   if ($u \notin mark$):
      mark.add($u$)
      for (($u,v$) $\in G$):
         put $v$ into $bag$
```

Definition 6.1 - *Depth-First Search*
A *Depth-First Search* of a graph explores to the end of a branch of vertices before backtracking.
A *Depth-First Search* can be implemented by implementing the *bag* in **Proposition 6.2** as a *stack.*[1]

[1]See `programs\Graph:depthFirstSearch`

Example 6.1 - *Depth-First Search*
Here we perform a *Depth-First Search* on the following tree.

```
        2
      /   \
   7        8
  / \       |
 3   6      9
    / \
   5 11
```

The nodes are met in the following order $2 \to 7 \to 3 \to 6 \to 5 \to 11 \to 8 \to 9$.

Definition 6.2 - *Breadth-First Search*
A *Breadth-First Search* of a graph meets all nodes at distance n from the origin before meeting any oat distance $n+1$.
A *Breadth-First Search* can be implemented by implementing the *bag* in **Proposition 6.2** as a *queue*.[1]
Example 6.2 - *Breadth-First Search*
Here we perform a *Breadth-First Search* on the following tree.

```
        2
      /   \
   7        8
  / \       |
 3   6      9
    / \
   5 11
```

The nodes are met in the following order $2 \to 7 \to 8 \to 3 \to 6 \to 9 \to 5 \to 11$.

3.7 Single Source Shortest Path

Definition 7.1 - *Single Source Shortest Path Problem*
Consider being given a graph & a starting vertex.
The *Single Source Shortest Path Problem* is to find the shortest path to all other vertices from the starting vertex.
N.B. - The length of a path can either be measured by the number of edges traversed or total sum of the weight of these edges.

Remark 7.1 - *Run-Time Complexity of Single Source Shortest Path Problem*
The *Run-Time Complexity* for the single source shortest path problem is the same as that for finding the shortest path between two given nodes, in the worst & average cases.
Since, you cannot be sure you have found a shortest path without checking all possible routes.

[1] See `programs\Graph:breadthFirstSearch`

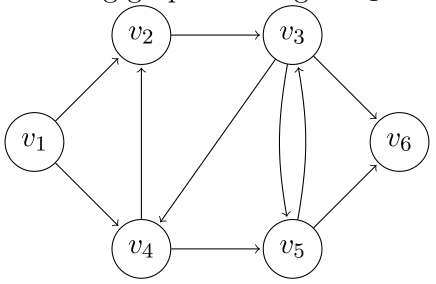
3.7.1 Unweighted Graph

Proposition 7.1 - *Breadth-First Search*
A *Breadth-First Search* can be used to solve the *Single Source Shortest Path Problem* for an unweigthted graph.
A *Breadth-First Search* to find the shortest path from s to all other vertices in G with the following pseudocode.[1]

```
BFS-SHORTEST-PATH(G, s)
mark=[]
q=QUEUE
for (v ∈ G):
    dist(v) = ∞
dist(s) = 0
q.put(s)
while (q !empty):
    take u from bag
    if (u ∉ mark):
        mark.add(u)
        for ((u,v) ∈ G):
            q.put(v)
            if (dist(v) == ∞):
                dist(v) = dist(u) + 1
return dist
```

Theorem 7.1 - *Run-Time Complexity for Breadth-First Search*
Breadth-First Search has run-time complexity of $\in O(|V|) + |E|)$ when solving the single source shortest path problem.

Example 7.1 - *Breadth-First Search*
Here we shall perform a *Breadth-First Search* to solve the *Single Source Shortest Path* for the following graph starting at v_1.



Set $dist = [v_1 : \infty, v_2 : \infty, v_3 : \infty, v_4 : \infty, v_5 : \infty, v_6 : \infty]$.
Set $dist[v_1] = 0 \implies dist = [v_1 : 0, v_2 : \infty, v_3 : \infty, v_4 : \infty, v_5 : \infty, v_6 : \infty]$.
Add v_1 to q. $q = [v_1]$.

[1] See `programs\SingleSourceShortestPath:breadthFirstSearch`

Take v_1 from q. $q = []$.
v_1 is unmarked. Mark v_1. mark=[v_1].
Add v_2 to q. $q = [v_2]$.
$dist[v_2] \equiv \infty$. Set $dist[v_2] = 0 + 1$
Add v_4 to q. $q = [v_2, v_4]$.
$dist[v_4] \equiv \infty$. Set $dist[v_4] = 0 + 1$
No more edges to check.
$dist = [v_1 : 0, v_2 : 1, v_3 : \infty, v_4 : 1, v_5 : \infty, v_6 : \infty]$.

Take v_2 from q. $q = [v_4]$.
v_2 is unmarked. Mark v_2. mark=[v_1, v_2].
Add v_3 to q. $q = [v_4, v_3]$.
$dist[v_3] \equiv \infty$. Set $dist[v_3] = 1 + 1$
No more edges to check.
$dist = [v_1 : 0, v_2 : 1, v_3 : 2, v_4 : 1, v_5 : \infty, v_6 : \infty]$.

Take v_4 from q. $q = [v_3]$.
v_4 is unmarked. Mark v_4. mark=[v_1, v_2, v_4].
Add v_2 to q. $q = [v_3, v_2]$.
$dist[v_3] \neq \infty$. Add v_5 to q. $q = [v_3, v_2, v_5]$.
$dist[v_5] \equiv \infty$. Set $dist[v_5] = 1 + 1$.
No more edges to check.
$dist = [v_1 : 0, v_2 : 1, v_3 : 2, v_4 : 1, v_5 : 2, v_6 : \infty]$.

Take v_3 from q. $q = [v_2, v_5]$.
v_3 is unmarked. Mark v_3. mark=[v_1, v_2, v_4, v_3].
Add v_4 to q. $q = [v_2, v_5, v_4, v_5]$.
$dist[v_4] \neq \infty$. Add v_5 to q. $q = [v_2, v_5, v_4, v_5]$.
$dist[v_5] \neq \infty$. Add v_6 to q. $q = [v_2, v_5, v_4, v_5, v_6]$.
$dist[v_6] \equiv \infty$. Set $dist[v_5] = 1 + 2$.
No more edges to check.
$dist = [v_1 : 0, v_2 : 1, v_3 : 2, v_4 : 1, v_5 : 2, v_6 : 3]$.

Take v_2 from q. $q = [v_5, v_4, v_5, v_6]$.
v_2 is marked.

Take v_5 from q. $q = [v_4, v_5, v_6]$.
v_5 is unarked. Mark v_5. mark=[v_1, v_2, v_4, v_3, v_5].
Add v_3 to q. $q = [v_5, v_6, v_3]$.
$dist[v_5] \neq \infty$.
Add v_6 to q. $q = [v_5, v_6, v_3, v_6]$.
$dist[v_6] \neq \infty$.
No more edges to check .

Take v_4 from q. $q = [v_5, v_6]$.
v_2 is marked.
Take v_5 from q. $q = [v_6]$.
v_2 is marked.

Take v_6 from q. $q = []$.
v_6 is unmarked. Mark v_6. mark=[v_1, v_2, v_4, v_3, v_5, v_6].

No more edges to check.

Done. $dist = [v_1 : 0, v_2 : 1, v_3 : 2, v_4 : 1, v_5 : 2, v_6 : 3]$.

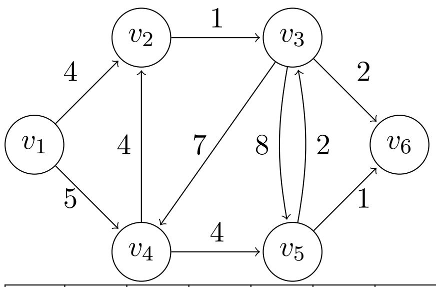
3.7.2 Weighted Graph

Definition 7.2 - *Dijkstra's Algorithm*
Dijkstra's Algorithm can be used to solve the *Single Source Shortest Path Problem* for a weighted graph.
Dijkstra's Algorithm requires all edges to have non-negative weightings and uses a priority queue[1].

Proposition 7.2 - *Pseudocode for Dijkstra's Algorithm*
Dijkstra's Algorithm can be performed on a vertex s and a weighted graph G using the following pseudocode

```
DIJSKTRA(G, s)
q=PRIORITY-QUEUE
For ($v \in G$):
   dist(v) = ∞
dist(s) = 0
for ($v \in G$):
  q.INSERT(v, dist(v))
while (q not empty):
  u = q.EXTRACT-MIN()
  for ($(u,v) \in G$):
  If (dist(v) > dist(u) + weight(u,v)):
     dist(v) = dist(u) + weight(u,v)
     q.DECREASE-KEY(v, dist(v))
return dist
```

Example 7.2 - *Dijkstra's Algorithm*
Here we shall perform *Dijkstra's Algorithm* on the follow graph, starting at v_1.



	v_1	v_2	v_3	v_4	v_5	v_6
	∞	∞	∞	∞	∞	∞
v_1	0	4	∞	5	∞	∞
v_2	0	4	5	5	∞	∞
v_4	0	4	5	5	9	∞
v_3	0	4	5	5	9	7
v_6	0	4	5	5	9	7
v_5	0	4	5	5	9	7

Priority Queue
$[(v_1,0),(v_2,\infty),(v_3,\infty),(v_4,\infty),(v_5,\infty),(v_6,\infty)]$
$[(v_2,4),(v_4,5),(v_3,\infty),(v_5,\infty),(v_6,\infty)]$
$[(v_4,5),(v_3,5),(v_5,\infty),(v_6,\infty)]$
$[(v_3,5),(v_5,9),(v_6,\infty)]$
$[(v_6,7),(v_5,9)]$
$[(v_5,9)]$
$[]$

[1]See **Subsection 2.2**

Theorem 7.2 - *Run-Time Complexity of Dijkstra's Algorithm*
The setup for *Dijkstra's Algorithm* has run-time complexity $\in O(|V|.T_{INSERT})$.
The *while* loop runs, at most, $|V|$ times.
The setup of each loop iteration has run-time complexity $\in O(T_{EXTRACT-MIN})$.
The *for* loop in the while loop runs, at most, $|E|$ times.
The process of each *for* loop has run-time complexity $\in O(T_{DECREASE-KEY})$.
Thus the *Run-Time Complexity of Dijkstra's Algorithm*

$$\in O(|V|.T_{INSERT} + |V|.T_{EXTRACT-MIN} + |V|.T_{DECREASE-KEY})$$

Remark 7.2 - *Summary of Run-Time Complexity for Dijkstra's Algorithm*
From **Theorem 7.2** we can see that the *Run-Time Complexity for Dijkstra's Algorithm* depends on what data structure is used to implement the *priority queue*.

	T_{INSERT}	$T_{DECREASE-KEY}$	$T_{EXTRACT-MIN}$	Dijkstra
Unsorted Linked List	O(1)	O($\|V\|$)	O($\|V\|$)	O($\|V\|^2 + \|V\|.\|E\|$)
Sorted Linked List	O($\|V\|$)	O($\|V\|$)	O(1)	O($\|V\|^2 + \|V\|.\|E\|$)
Binary Heap	O($\log_2 \|V\|$)	O($\log_2 \|V\|$)	O($\log_2 \|V\|$)	O(($\|V\| + \|E\|) \log_2 \|V\|$)
Finomacci Heap	O(1)	O(1)	O($\log_2 \|V\|$)	O($\|E\| + \|V\| \log_2 \|V\|$)

Proof 7.1 - *Correctness of Dijkstra's Algorithm*[1]
Here we prove that once Dijkstra's Algorithm has terminated, $dist(v)$ is the shortest path $s \to v$.
This is a proof by contradiction.
Let $\delta(s, v)$ denote the true value of the shortest path from s to v.
Let v be the first vertex to be extracted where $dist(v) \neq \delta(s, v)$.
Then $v \neq s$ since the value of $dist(s)$ is hard-coded to be correct in the algorithm.
And, there must be a path $s \to v$ otherwise $dist(v) = \infty = \delta(s, v)$.
Consider the state of the algorithm directly before v is extracted.
We have that v is in queue, but s is not.
Set y to be the first vertex on the path $s \to v$ which is still in the queue.
Set x to be the point before y, on the path $s \to v$ which is not in the queue (x may equal s).
We know $s \to y$ is a shortest path by the condition that v is the first vertex to be extracted which is not a shortest path.
Thus, $\delta(s, y) \leq \delta(s, v)$.
Since vertex x has been extracted, we have $dist(x) = \delta(s, x)$ by the same condition.
When we extracted x we relaxed the edge (x, y).

$$\begin{array}{rrcl} \implies & dist(y) \leq \delta(s,x) + weight(x,y) & = & \delta(s,y) \\ \implies & dist(y) \leq \delta(s,y) & \& & \delta(s,y) \leq \delta(s,v) \end{array}$$

Since v is extracted before y, $dist(v) \leq dist(y)$.

$$\begin{array}{rrcl} \implies & dist(v) & \leq & dist(y) \leq \delta(s,y) \leq \delta(s,v) \\ \implies & dist(v) & \leq & \delta(s,v) \end{array}$$

However, since v is in the queue $dist(v) \geq \delta(s, v)$.
Thus $dist(v) = \delta(s, v)$.
This is a contradiction of the conditions.
Thus, due to the contraction, the claim holds.

[1]This is non-examinable

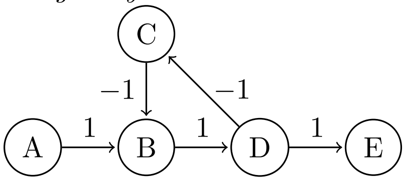
Definition 7.3 - *Relaxing an Edge*
Relaxing an edge is the process of trying to lower the cost of getting to a vertex, by using another vertex.
N.B. - This happens in the last three lines of the pseudocode in **Proposition 7.2**.

Definition 7.4 - *Settled Vertices*
A vertex is said to be *settled* once the shortest path to it has been found.
N.B. - In the pseudocode in **Proposition 7.2** in this happens when EXTRACT-MIN() is run.

3.7.3 Negative Weighted Graph

Definition 7.5 - *Negative Weight Cycle*
A *Negative Weight Cycle* is a cycle in a graph ($A \to \cdots \to A$) where the sum of weights is negative.
By repeatdly going around this cycle the weight of a path will keep decreasing.
This means shortest source algirthms never terminate.

Example 7.3 - *Negative Weight Cycle*
In the following graph the cycle $B \to D \to C \to B$ has a total weight of -1 & is a *Negative Weight Cycle*.

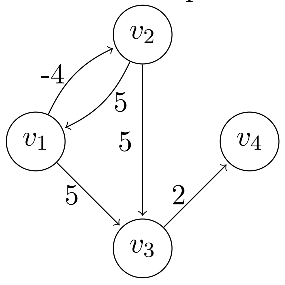


Definition 7.6 - *Bellman-Ford Algorithm*
The *Bellman-Ford Algorthm* solves the single source shortest path problem for graphs with negative weights.
It is made of two parts: The first finds the cheapest cost from a source to every other node, and the second identifies any negative weight cycles and discards them.
Bellman-Ford Algorithm can be applied to a graph G and source s using the following pseudocode

```
BELLMAN-FORD(G, s)
for (v∈G):
  dist(v)=∞
dist(s)=0
for (i=0; i<|V|; i++):[1]
  for ((u,v)∈ E):
    if (dist(v)>dist(u)+weight(u,v)): [2]
      dist(v)=dist(u)+weight(u,v)
for ((u,v)∈ E):
  if (dist(v)>dist(u)+weight(u,v)):
    return ''Negative weight cycle found"
return dist
```

[1] i is a dummy variable; [2] this is relaxing edge (u,v).

Example 7.4 - *Bellman-Ford Algorithm*
Here we shall perform the *Bellman-Ford Algorithm* on the follow graph, starting at v_1.



We will analyse the edges in the following order (v_1, v_2), (v_1, v_3), (v_2, v_1), (v_2, v_3), (v_3, v_4).
Since there are 5 edges we shall do 5 passes.

Pass 1.
dist=[$(v_1, 0)$,(v_2, ∞),(v_3, ∞),(v_4, ∞)].
Analysing (v_1, v_2). $0 - 4 = -4 < \infty$ update $dist(v_2) = -4$.
Analysing (v_1, v_3). $0 + 5 = 5 < \infty$ update $(v_3) = 5$.
Analysing (v_2, v_1). $-4 + 5 = 1 \geq 0$ do nothing.
Analysing (v_2, v_3). $-4 + 5 = 1 < 5$ update $(v_3) = 1$.
Analysing (v_3, v_4). $1 + 2 = 3 < \infty$ update $(v_4) = 3$.

Pass 2.
dist=[$(v_1, 0)$,$(v_2, -4)$,$(v_3, 1)$,$(v_4, 3)$].
Analysing (v_1, v_2). $0 - 4 = -4 \geq -4$ do nothing.
Analysing (v_1, v_3). $0 + 5 = 5 \geq 1$ do nothing.
Analysing (v_2, v_1). $-4 + 5 = 1 \geq 0$ do nothing.
Analysing (v_2, v_3). $-4 + 5 = 1 \geq 1$ do nothing.
Analysing (v_3, v_4). $1 + 2 = 3 \geq 3$ do nothing

Since there were no updates in Pass 2 there will be none in passes 3,4 & 5.

Now checking for negative weight cycles.
Analysing (v_1, v_2). $0 - 4 = -4 \geq -4$ do nothing.
Analysing (v_1, v_3). $0 + 5 = 5 \geq 1$ do nothing.
Analysing (v_2, v_1). $-4 + 5 = 1 \geq 0$ do nothing.
Analysing (v_2, v_3). $-4 + 5 = 1 \geq 1$ do nothing.
Analysing (v_3, v_4). $1 + 2 = 3 \geq 3$ do nothing

return [$(v_1, 0)$,$(v_2, -4)$,$(v_3, 1)$,$(v_4, 3)$].

Theorem 7.3 - *Run-Time Complexity of Bellman-Ford Algorithm*
Setting the initial values has run-time complexity $\in O(|V|)$.
Finding the shortest path has run-time complexity $\in O(|V||E|)$.
Finding the negative weight cycles has run-time complexity $\in O(|E|)$.
Thus the overall run-time complexity of $BELLMAN - FORD \in O(|V||E|)$.

Proof 7.2 - *Correctness of Bellman-Ford Algorithm - 1* [1]
Here we prove that if there are no negative weight cycles in the graph then the shortest path $s \to t$ contains at most $|V|$ edges, or there is no path $s \to t$.
Proof.
Consider a path $s \to t$ which contains a cycle.
Since the graph contains no negative weight cycles then removing the cycle from the path will not increase the total weight of the path or the number of edges.
Thus the shortest path contains no cycles and so can only have at most $|V|$ edges.

Proof 7.3 - *Correctness of Bellman-Ford Algorithm - 2*[2]
Here we prove that when the algorithm terminates, for each vertex v $dist(v)$ is the length of the shortest path between s and v.
This is a proof by.
Suppose that each iteration we relax just one edge, rather than relax all edges.
Further, suppose the edge we relax on the i^{th} iteration is the i^{th} edge in a shortest path from s to some vertex t.
Then when the algorithm terminates, $dist(t)$ is the length of the shortest path $s \to t$.
In the proper algorithm where we relax every edge, at some point of the i^{th} iteration we relax the i^{th} edge in the shortest path from $s \to t$.
The maximum length of a path is $|V|$.
Then $|V|$ iterations are sufficient to guarantee the shortest path is found.
Since the algorithm always does $|V|$ iterations then we can use the conclusion from **Proof 7.2** to show the claim is valid.

Proof 7.4 - *The Bellman-Ford Algorithm Detects Negative Weight Cycles* [3]
Here we prove that the third *for* loop of the *Bellman-Ford Algorithm* does detect negative weight cycles correctly.
This is a proof by contradiction.
We know that after the first part of the algorithm $dist(v)$ is the length of the shortest path $s \to v$.
If the final check is triggered then $dist(u) + weight(u, v) < dist(v)$.
This has found a path shorter than the shortest path, which is a contradiction.
Let $v_1, v_2, \dots, v_k \in V$ be a negative weight cycle that wasn't reported by the *Bellman-Ford Algorithm.*
$weight(v_1, v_2) \geq dist(v_2) - dist(v_1)$ since the check that $dist(v_2) > dist(v_1) + weight(v_1, v_2)$ wasn't triggered.
Then $dist(v_2) \leq dist(v_1) + weight(v_1, v_2)$.
By the same argument $weight(v_2, v_3) \geq dist(v_3) - dist(v_2), \dots$
We can generalise this to $weight(v_j, v_{j+1}) \geq dist(v_{j+1}) - dist(v_j)\ \forall\ (v_j, v_{j+1})$ Set $v_{k+1} = v_1$.
Since this cycle has negative weight then $\sum_{j=1}^{k} weight(v_j, v_{j+1}) < 0$.
However $\sum_{j=1}^{k} weight(v_j, v_{j+1}) \geq \sum_{j=1}^{k} dist(v_{j+1}) - \sum_{j=1}^{k} dist(v_j)$.
These two inequalities form a contradiction.
This is proof of the claim.

[1]This is non-examinable
[2]This is non-examinable
[3]This is non-examinable

3.8 All Pairs Shortest Path

Definition 8.1 - *All Pairs Shortest Path Problem*
Consider being given a graph.
The *All Pairs Shortest Path Problem* is to find the shortest path between all pairs of vertices.

Proposition 8.1 - *Naïve Approach*
For a non-negative weighted graph we can run *Dijkstra's Algorithm*[1] on every vertex.
This approach has run-time complexity $\in O(|V|(|V|+|E|)\log_2|V|) \equiv O(|V||E|\log_2|V|)$ when implementing the priority queue as a binary heap[2].
For a negatively weighted graph we can run the *Bellman-Ford Algorithm*[3] on every vertex. This approach has run-time complexity $\in O(|V|^2|E|)$[4].

Definition 8.2 - *Dense & Sparse Graphs*
If $|E| \approx |V|^2$ then a graph is said to be *Dense*.
If $|E| \approx |V|$ then a graph is said to be *Sparse*.
N.B. - The maximum number of edges is $|V|(|V|-1)$.

Proposition 8.2 - *Reweighting a Graph with a constant*
Adding a constant to every edge in a graph disproportionately increases the total cost of paths with more edges, compared to shorter paths.

Proposition 8.3 - *Finding Potential*
The following process can be used to find a potential for all vertices.

1. Add a new vertex s to the graph.
2. Create a zero-weighted edge $(s,v)\ \forall\ v\in V$.
3. Let $\delta(s,v) = BELLMAN-FORD(G,s)$. The values this maps to will be ≤ 0. If there are any negative weight cycles then $\delta(s,v)$ may be indefined.
4. Define $h(v)=\delta(s,v)\ \forall\ v\in V$.

N.B. - When implementing remember to remove s and all edges connected to it.

Proposition 8.4 - *Reweighting a Graph*
Here is a technique for rewighting a graph in order to make all edges non-negatively weighted.
Use **Proposition 8.3** to find $h(v)$.
Update the weight of all edges so that

$$weight'(u,v) = weight(u,v) + h(u) - h(v)$$

[1]See **Definition 7.2**
[2]See **Remark 7.2**
[3]See **Definition 7.6**
[4]See **Theorem 7.3**

Remark 8.1 - *Effect of Reweighting on a Path*
Consider a path $x \to y \to z$.
Before reweighting the total weight of this path is $w(x, y) + w(y, z)$.
After reweight the total weight of this path is $(w(x, y) + h(x) - h(y)) + (w(y, z) + h(y) - h(z))$.
Notice that the potentials for y cancel out, given a final reweighted value of $w(x, y) + w(y, z) + h(x) - h(z)$.
Thus the reweighted value is simple the original weight of the path plus the potential of the origin, less the potential of the terminal.
This can be extended to paths of any length.

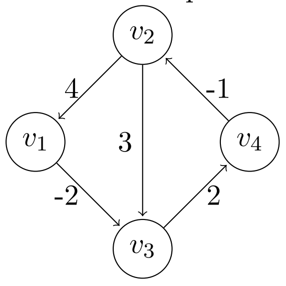
3.9 Negatively Weighted Graph

Definition 9.1 - *Johnson's Algorithm*
Johnson's Algorithm solves the all pairs shortest path problem for graphs with both positively & negatively weighted edges.
Johnson's Algorithm reweights a graph so it has no negative weights and then uses both *Dijkstra's Algorithm*[1] & the *Bellman-Ford Algorithm*[2].
Johnson's Algorithm can be performed on graph G using the following pseudocode

```
JOHNSON(G)
G.addVertex(s)
for (v∈G):
  G.addEdge(s,v,0)
δ=BELLMAN-FORD(G,s)
for ((u,v)∈G):
  G.weight(u,v)=G.weight(u,v)+δ(s,u)−δ(s,v)
for (v∈G):
  dists.add(DIJKSTRA(G,v))
for (u∈G):
  for (v∈G):
    dists(u,v)=dists(u,v)+δ(s,v)−δ(s,u)
return dists
```

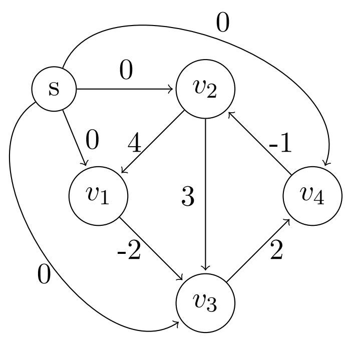
N.B. - We deweight all edges before returning.

Example 9.1 - *Johnson's Algorithm*
Here we shall perform *Johnson's Algoritm* on the following graph

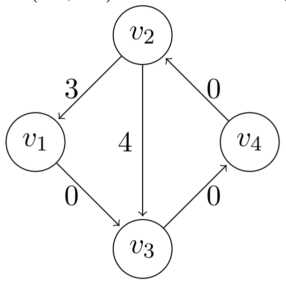


Add vertex s and zero-weighted edges to all other vertices.

[1]See **Definition 7.2**
[2]See **Definition 7.6**



δ=BELLMAN-FORD(G,s)=[(v_1,0),(v_2,-1),(v_3,-2),(v_4,0)].
Reweight graph with
$w(v_1, v_3) = -2 + 0 - (-2) = 0.$
$w(v_2, v_1) = 4 + (-1) - 0 = 3.$
$w(v_2, v_3) = 3 + (-1) - (-2) = 4.$
$w(v_3, v_4) = 2 + (-2) - 0 = 0.$
$w(v_4, v_2) = -1 + 0 - (-1) = 0.$



Running *Dijkstra's Algorithm* on all vertices produces the following table.

	v_1	v_2	v_3	v_4
v_1	0	0	0	0
v_2	3	0	3	3
v_3	3	0	0	0
v_4	3	0	3	0

Deweighting these values we produce the following table.

	v_1	v_2	v_3	v_4
v_1	0+0-0	0+(-1)-0	0+(-2)+0	0+0-0
v_2	3+0-(-1)	0+(-1)-(-1)	3+(-2)-(-1)	3+0-(-1)
v_3	3+0-(-2)	0+(-1)-(-2)	0+(-2)-(-2)	0+0-(-2)
v_4	3+0-0	0+(-1)-0	3+(-2)-0	0+0-0

This gives a final table of results as

	v_1	v_2	v_3	v_4
v_1	0	-1	-2	0
v_2	4	0	2	4
v_3	5	1	0	2
v_4	3	-1	1	0

N.B. - The values in these tables are the cheapest cost from row element to column element.

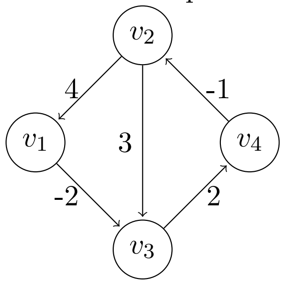
Theorem 9.1 - *Run-Time Complexity of Johnson's Algorithm*
Add the new vertex has run-time complexity $\in O(1)$.
Creating the new edges has run-time complexity $\in O(|V|)$.
Running the *Bellman-Ford Algorithm* on s has run-time complexity $\in O(|V||E|)$.
Rewighting the graph has run-time complexity $\in O(|E|)$.

Running *Dijkstra's Algorithm* on all edges has run-time complexity $\in O(|V||E|\log_2|V|)$.
And deweighting the values has run-time complexity $\in O(|V|^2)$.
Thus the overall run-time complexity for *Johnson's Algorithm* $\in O(|V||E|\log_2|V|)$.
N.B. - If the graph is sparse, with $|E| \leq |V|/2$ then the run-time complexity $\in O(|V|^2 + |V||E|\log_2|V|)$.

Definition 9.2 - *Floyd-Warshall Algorithm*
The *Floyd-Warshall Algorithm* solves the all pairs shortest path problem for graphs with both positively & negatively weighted edges.
The *Floyd-Warshall Algorithm* considers paths between adjacent vertex.
Then, with each iteration it allows the paths to go through one extra specified vertex.
The *Floyd-Warshall Algorithm* can be performed on graph G using the following pseudocode

```
FLOYD-WARSHALL(G)
for ((u,v)∈G):
  dists(u,v)=weight(u,v)
for (v∈G):
  dists(v,v)=0
for (k∈G):
  for (i∈G):
    for (j∈G):
      if (dists(i,j)>dists(i,k)+dists(k,j)):
        dists(i,j)=dists(i,k)+dists(k,j)
return dists
```

Example 9.2 - *Floyd-Warshall Algorithm*
Here we shall perform the *Floyds-Warshall Algoritm* on the following graph



k=0. Only allow paths of length 1.
$v_1 \to v_3 = -2$.
$v_2 \to v_1 = 4$.
$v_2 \to v_3 = 3$.
$v_3 \to v_4 = 2$.
$v_4 \to v_2 = -1$.
k=1. Allow paths to use $[v_1]$.
$v_2 \to v_1 \to v_3 = 2$. Update.
k=2. Allow paths to use $[v_1, v_2]$.
$v_4 \to v_2 \to v_1 = 2$.
$v_4 \to v_2 \to v_1 \to v_3 = 1$.
k=3. Allow paths to use $[v_1, v_2, v_3]$.
$v_1 \to v_3 \to v_4 = 0$.
$v_2 \to v_1 \to v_3 \to v_4 = 4$.
k=4. Allow paths to use $[v_1, v_2, v_3, v_4]$.
$v_3 \to v_4 \to v_2 = 1$.
$v_3 \to v_4 \to v_2 \to v_1 = 5$.
$v_1 \to v_3 \to v_4 \to v_2 = -1$.

Returns

	v_1	v_2	v_3	v_4
v_1	0	-1	-2	0
v_2	4	0	2	4
v_3	5	1	0	2
v_4	2	-1	1	0

Theorem 9.2 - *Run-Time Complexity of Floyd-Warshall Algorithm*
To find all $|V|^2$ values of $shortestPath(i,j,k)$ from the values of $shortestPath(i,j,k-1)$ requires $2|V|^2$ operations.
Since we compute $shortestPath(i,j,k)$ $|V|$ times the total number of operations is $2|V|^3$.
Thus the run-time complexity for the *Floyd-Warshall Algorithm* $\in \Theta(n^3)$.

3.10 Minimum Spanning Trees

Definition 10.1 - *Kruskal's Algorithm*
Kruskal's Algorithm finds a minimum spanning tree[1] using a disjoint set[2] with elements $\{1,2,\dots,|V|\}$.

i) $\forall\ v \in V$ do $MAKE-SET(v)$ ($\in O(|V|)$).

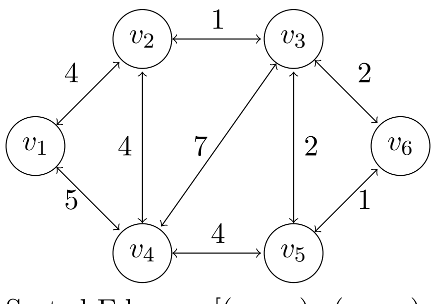
ii) Sort the edges in increasing order of weight ($\in O(|E|\log|E|)$.

iii) $\forall (u,v) \in E$, in order, if $FIND-SET(u) \neq FIND-SET(v)$ then $UNION(u,v)$ and add $(u,v,)$ to tree ($\in O(\log|V|)$)

Proposition 10.1 - *Pseudocode for Kruskal's Algorithm*
Kruskal's Algorithm can be used to find a minimum spanning tree in a graph G using the following pseudocode.[3]

```
KRUSKAL(G)
tree=GRAPH
sets=DISJOINT-SETS
sorted=SORT($G.edges$)
for ($v \in G$):
   tree.add(v)
   sets.MAKE-SET(v)
for ($(u,v) \in sorted$):
   if (sets.FIND-SET(u)!=sets.FIND-SET(v)):
      sets.UNION(u,v)
      tree.add((u,v))
return tree
```

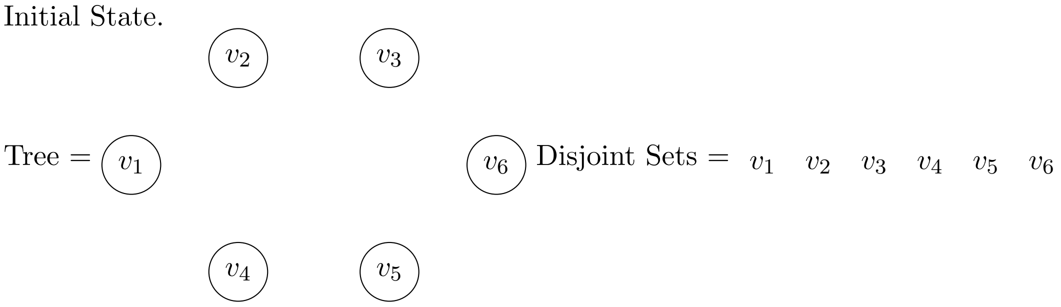
Example 10.1 - *Kruskal's Algorithm*
Here we shall perform *Kruskal's Algorithm* on the follow graph.



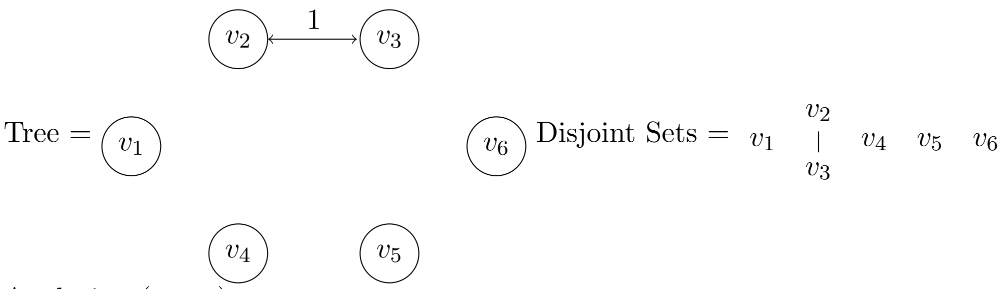
Sorted Edges = $[(v_2,v_3),\ (v_5,v_6),\ (v_3,v_6),\ (v_3,v_5),\ (v_1,v_2),\ (v_2,v_4),\ (v_4,v_5),\ (v_1,v_4),\ (v_3,v_4)]$.

[1]See **Data Structures:Definition 4.3**
[2]See **Subsection 2.4**
[3]See `programs\Graph.java:kruskal`

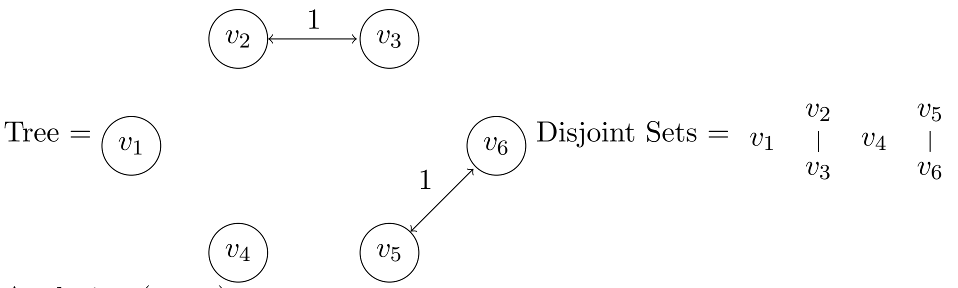
Initial State.



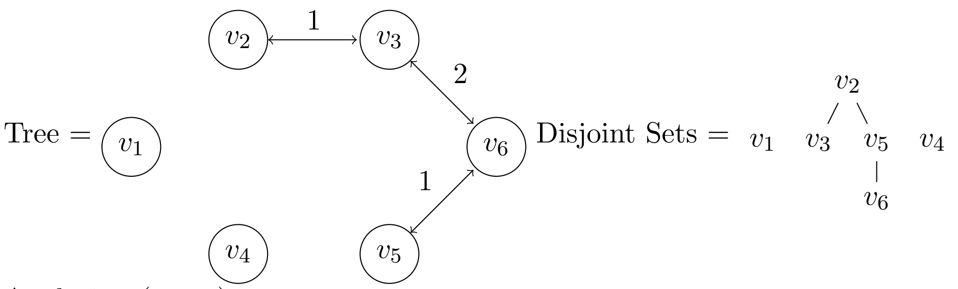
Analysing (v_2, v_3).



Analysing (v_5, v_6).

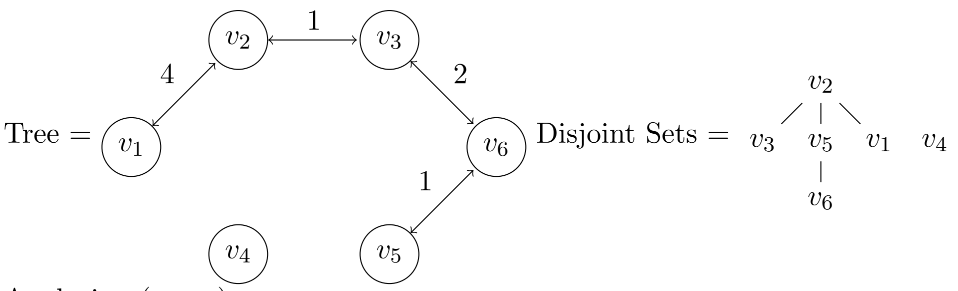


Analysing (v_3, v_6).

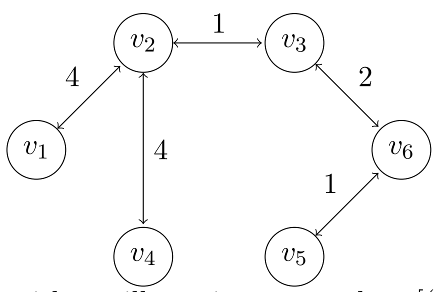


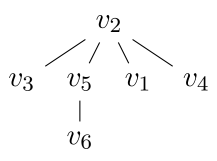
Analysing (v_3, v_5).
v_3 & v_5 are already in the same set.

Analysing (v_1, v_2).



Analysing (v_2, v_4).

Tree =


Disjoint Sets =


The algorithm will continue to analyse $[(v_4, v_5),\ (v_1, v_4),\ (v_3, v_4)]$ but these will have no effect as they are all in the same set.

Theorem 10.1 - *Run-Time Complexity of Kruskal's Algorithm*
Kruskal's Algorithm has run-time complexity $\in O(|V| \log |V|)$ since $log_{|}E| \leq \log |V|$.

Proof 10.1 - *Sketch Correctness of Kruskal's Algorithm*
Here we prove that the spanning tree output by *Kruskal's Algorithm* is a minimum spanning tree for the given graph.
This is a sketch proof.
Let K be the spanning tree outputted by *Kruskal's Algorithm.*
Let M be any minimum spanning tree such that $M \neq K$.
Let e be the edge with lowest weight that is in K but not in M.
If we add e to M we would produce a cycle since M is a spanning tree.
There must be an edge f in this potential cycle which in not in K, since K contains no cycles.
Let M_2 be M with e added and f removed.
M_2 is another minimum spanning tree, with one more edge in common with K than M.
The proof that K is a minimum spanning tree then follows from repeatedly applying the argument.
N.B. - If there is a minimum spanning tree with 1 edge in common with K, then there is one with 2 edges in common, $\dots$ then there is one with all n edges in common.

3.11 Dynamic Programming

Definition 11.1 - *Dynamic Programming*
Dynamic Programming is an approach to algorithmic design.
It is a technique for finding efficient algorithms for problems which can be broken down into smaller, overlapping problems.

Proposition 11.1 - *Process of Dynamic Programming*
Here is an overview of the process of *Dynamic Programming.*

i) Find a recursive formula for the problem, in terms of answers to the sub-problems.

ii) Write down the naive recursive problem.

iii) Speed it up by storing the solution to sub-problems.

iv) Derive an iterative algorithm by solving the sub-problems in a good order.

Definition 11.2 - *Memoisation*
Memoisation is optimisation technique where you store calculated values so they can be read at a later stage, rather than having to recalculate.

Definition 11.3 - *Dependency Graph*
A *Dependency Graph* is a directed graph where the children of each node represent the values it depends upon.

3.11.1 Largest Empty Square Problem

Definition 11.4 - *Largest Empty Square Problem*
Consider having a picture containing white & black pixels.
The *Largest Empty Square Problem* is to find the largest square of only white pixels.
This problem can be solved by dynamic programming.
N.B. - A square of only white pixels is referred to as empty.

Proposition 11.2 - *Finding Solution to Largest Empty Square Problem*

i) *Recursive Formula.*
Consider the fact that any $m \times m$ square of pixels, S, is empty if and only if the bottom right pixel of S is empty and the three $(m-1)\times(m-1)$ squares in the top-left, top-right & bottom-left corners are empty.
Let $LES(x,y)$ be the side length of the largest empty square whose bottom right pixel is at (x,y). Then

   (a) If (x,y) is not empty then $LES(x,y)=0$.
   (b) Else if (x,y) is in the first row of first column then $LES(x,y)=1$.
   (c) Else $LES(x,y)=min\{LES(x-1,y-1),LES(x-1,y),LES(x,y-1)\}+1$.

   N.B. - We know $LES(x,y)\leq LES(x-1,y-1)+1$, $LES(x,y)\leq LES(x-1,y)+1$ & $LES(x,y)\leq LES(x,y-1)+1$.

ii) *Code for Recursive Formula.* This recursive method can be implemented by the following pseudocode

```
LES(x,y)
if ((x,y) !empty):
  return 0
if ((x≡1)||(y≡1)):
  return 1
return (min{LES(x−1,y−1), LES(x−1,y), LES(x,y−1)}+1)
```

   This code is slow & inefficient due to many overlapping calculations caused by the recursion.
   It will run in approximately $O(3^n)$.
   By considering the tree formed by the recursion we see where the overlaps occur as subtrees.

iii) *Store Solutions of Subproblems.*
We can use an array to store values, rather than recalculating them.
The first time we want a value for a position, we shall calculate it and store it in the array.
On all subsequent occasions we shall just read it from the array.
This technique can be implemented by the following pseudocode

```
MEM-LES(x,y)
if ((x,y) !empty):
  return 0
if ((x≡1)||(y≡1)):
  return 1
if (LES[x,y]≡NULL):
  LES[x,y]=min{LES(x-1,y-1), LES(x-1,y), LES(x,y-1)}+1
return LES[x,y]
```

N.B. - This technique has run-time complexity$\in O(n^2)$.

iv) *Derive an Iterative Algorithm* Here we need consider in what order to calculate values so that the values a position is dependent upon are calculated before that position.
For this problem we need to calculate the values for $(x-1,y-1)$, $(x-1,y)$ & $(x,y-1)$ before calculating the value for (x,y).
Thus, solving values left-to-right top-to-bottom is valid.
This can be implemented by the following pseudocode

```
IT-LES(n)
for (y=1; y≤n; y++):
  for (x=1; x≤n; x++):
    if ((x,y) !empty):
      LES[x,y]=0
    else if ((x≡1)||(y≡1)):
      LES[x,y]=1
    else:
      LES[x,y]=min{LES[x-1,y-1], LES[x-1,y], LES[x,y-1]}+1
return LES
```

N.B. - $IT-LES(n)$ calculates all values for the positions up to (n,n).

3.11.2 Weighted Interval Problem

Definition 11.5 - *Weighted Interval Problem*
Consider being given an array of weighted intervals which are ordered by their finishing times.
The intervals are given in triples st $A[i]=s_i,f_i,w_i$.
The *Weighted Interval Problem* is to find the schedule of greatest total weight within A.
This can be solved by dynamic programming.

Definition 11.6 - *Intervals*
An *Interval* has a start time & end time.
A *Weighted Interval* has a weight, start time & end time.

Definition 11.7 - *Compatible Intervals*
Two intervals are said to be *Compatible* if they do not overlap in terms of time.

Definition 11.8 - *Schedule*
A *Schedule* is a sequence of compatible intervals.

Proposition 11.3 - *Finding Solution to Weighted Interval Problem*
let A be the array of intervals we are trying to find the schedule of greatest total weight within.
Let p_i be the rightmost compatible interval to i, which finished before i.
Set $p_i = 0$ if no such interval exists.

i) *Recursive Formula.*
Let O be the optimal schedule and O_w be the weight of O.
Consider the n^{th} element of A.
Either it is in O or not.
In the case that the n^{th} element is *not in* O we have that O is the same for $a[1, \dots, n-1]$ and $O_w(n) = O_w(n-1)$.
In the case that the n^{th} element is *in* O we know no intervals that overlap with it are in O so we now only need to consider $\{1, \dots, p(n)\}$ and we know $O_w(n) = O_w(p(n)) + w_n$.
We can tell which case to consider by the maximum value of $O_w(n-1)$ & $O_w(p(n)) + w_n$.

ii) *Code for Recursive Formula.*
This recursive method can be implemented by the following pseudocode

```
WIS(i)
 if (i==0)
   return 0
 return max(WIS(i-1), WIS($p_i$)+$w_i$)
```

This returns the maximum weight for a sequence of the intervals $\{1, \dots, i\}$.
N.B. - We have that $T(n) \geq 2T(n-2)$ so this has run-time complexity$\in O(2^n)$.

iii) *Store Solutions of Subproblems.*

```
MEM-WIS(i)
 if (i==0)
   return 0
 if (WIS[i]==null)
   WIS[i]=max(MEM-WIS(i-1), MEM-WIS($p_i$)+$w_i$)
```

N.B. - Since we only need to calculate each value once, this has run-time complexity $\in O(n)$.

iv) *Derive an Iterative Algorithm.*
The value $WIS[i]$ depends upon $WIS[i-1]$ and $WIS[p_i]$ (which might be the same).
Since both these values lie to the left of $WIS[i]$ in the array WIS we can see that an iterative solution is possible, by calculating values from left-to-right.

```
IT-WIS(n)
 if (n≡0)
   return 0
 for (i=1; i++;i <n+1)
   WIS[i]=max(WIS[i-1],WIS[$p_i$]+$w_i$)
 return WIS[n]
```

N.B. - This has run-time complexity $\in O(n)$.

Proposition 11.4 - *Decomposing Sequence from Value*
Using the process defined in **Proposition 10.3** we can calculate the array WIS where $WIS[i]$ is the maximum weight of a sequence $\{1, \dots, i\}$.
The following pseudocode can be used to work out the composition of said sequence

```
FIND–WIS(i)
if (i≡0)
  return null
if (WIS[i] <= WIS[$p_i$]+w_i)
  return FIND–WIS($p_i$) then i
return FIND–WIS(i-1)
```

N.B. - This has time complexity $O(n)$.

Remark 11.1 - *Finding p_is*
Suppose we want to find the value of p_i for an interval i.
We want to find interval j such that $f_j \leq s_i \leq f_{j+1}$.
Since the given array is sorted by finish time this can be done by a binary search.
This has run-time complexity $\in O(\log_2 n)$.
Thus the run-time complexity to calculate all n of the p_is $\in O(n \log_2 n)$.

Proposition 11.5 - *Time Complexity to Solve Weighted Interval Problem*
Pre-computation of p_is has run-time complexity $\in O(n \log_2 n)$.
Computing WIS has run-time complexity $\in O(n)$.
Computing the schedule from WIS has run-time complexity $\in O(n)$.
Thus the run-time complexity to solve the whole problem $\in O(n \log_2 n)$.

3.12 Line Intersection Problem

Definition 12.1 - *Line Intersection Problem*
Consider being given the co-ordinates for the two end points to a series of lines.
The *Line Intersection Problem* is to find all the intersections between these lines.

Proposition 12.1 - *Naïve Approach*
The *Naïve Approach* may be to check if any lines intersect with any other line.
This can be done by the following pseudocode

```
NAIVE–INTERSECTION(n, s)
for (i=1; i≤n; i++):
  for (j=1; j≤n; j++):
    if (($s_i$ intersects $s_j$) && ($i != j$)):
      return (i,j)
```

Checking for an intersection has run-time complexity $\in O(n)$.
Thus, this method has run-time complexity $\in O(n^2)$.

Remark 12.1 - *Number of Intersections*
In the worst case scenario every line intersects with every other line, so there are $n(n-1)$ different intersections.
This means space complexity is $\in O(n^2)$ and any algorithm has run-time complexity $\Omega(n^2)$.

Remark 12.2 - *Solution Restrictions*
The solution that is about to be defined only works on an input with the following restrictions

- No horizontal lines;
- No overlapping lines;
- No shared end points;
- No points where there lines intersect at once.

Proposition 12.2 - *Reducing Restrictions*
To remove these restrictions on the input we could add random noise to all end points.
This however doesn't guarantee these conflicts disappear.
Alternatively we could preprocess the horizontal lines, merge the overlapping lines, split shared end points by slightly increasing one of their x values, and when an intersection of three lines occurs invert the order rather than swap.

Definition 12.2 - *Y-Span*
The *Y-Span* of a line is the range of the y values the line covers.

Proposition 12.3 - *Y-Span & Intersection*
If two lines don't have overlapping *y-span*s then they can never intersect.

Definition 12.3 - *Adjacent*
Two lines segments, s_1 & s_2, are considered adjacent at a given y value if no other line crosses that y line between s_1 & s_2.
This means two line segments can be adjacent for one y value and not another.
N.B. - If two lines are never adjacent then they never intersect.

Definition 12.4 - *Sweep Line*
The *Sweep Line* is a horizontal line which passes down the plane containing the line segments.
The *Status* of the *Sweep Line* is the set of lines which intersect the sweep line, ordered left to right.

Definition 12.5 - *Event Points*
Event Points are co-ordinates where the status of the sweep line changes.
For the *Line Intersection Problem* these are all the line end points & intersection points.

Proposition 12.4 - *Good Solution to Line Intersection Problem*
Here we define a better method for solving the *Line Intersection Problem.*
Setup
Define a *sweep line* & a collection of *event points.*
Create an empty data structure for the *status* of the sweep line.
Process
While the collection of *event points* is not empty, perform the following analysis on all the top event point.

i) If event point is top of a line segment, insert into status.
Check if this line intersects with either of its adjacent segments in the status.
If it does then add new event points to the collection.

ii) If event point is bottom of a line segment, delete from status.

iii) If event point is an intersection point, swap the order of those two line segments in the status.

Proposition 12.5 - *Data Structures to for Proposition 11.4*
The event points can be tracked by a priority queue.
The status can be stored by a self-balancing tree, using the end points as the key.

Proof 12.1 - *Correctness of Proposition 11.4*
Here we prove it is not possible for the method in **Proposition 10.4** to miss a solution.
Suppose two lines intersect, then they must become adjacent at some point.
This can only occur at an event point.
Since we find all end points, we will find all the intersections.

3.13 Linear Programming

Definition 13.1 - *Linear Program*
A *Linear Program* has is a linear function[1] which is subject to a set of linear inequalities[2].
N.B. - The linear function is called the *Objective Function.*

Definition 13.2 - *Linear Programming Problem*
A *Linear Programming Problem* requires you to optimise the objective function of a given linear program.
This can be either to maximise or, minimise the objective function.

Definition 13.3 - *Simplex Algorithm*
The *Simplex Algorithm* is an algorithm for solving linear programming problems.
The *Simplex Algorithm* takes in a linear program & returns the optimal solution.
The *Simplex Algorithm* starts at a vertex in the feasible region and performs a sequence of iterations.
At each iteration the *Simplex Algorithm* moves along an edge to a neighbouring vertex whose objective value is better than the current vertex.
It terminates when it reaches a local optimum.

Definition 13.4 - *Non-Standard Linear Program*
A linear program is in *Non-Standard Form* if

i) The problem is to minimise, rather than maximise, the objective function;

ii) There are varaibles without non-negative constraints (≥ 0);

iii) There are constraints with equality, rather than inequalities;

iv) Or, there are inequality constraints which need to change $\geq$, not $\leq$.

Proposition 13.1 - *Converting to Standard Form*
A non-standard program can be converted to standard form by

i) If minimising, negate all coefficients of the objective function.

ii) If the a variable y doesn't have a non-negative constraint set $y = y_1 - y_2$ and $y_1, y_2 \geq 0$.

iii) If a constraint in an equality, swap for two inequality versions. One $\leq$ and one $\geq$.

iv) To change from $\geq$ to $\leq$ negate all coefficients.

[1]See **Reference:Definition 3.5**
[2]See **Reference:Definition 3.6**

Example 13.1 - *Converting to Standard Form*
Non-standard form

$$\begin{array}{rrcl} \text{Minimise} & 2x_1 + 7x_2 + x_3 & & \\ \text{Subject to} & x_1 - x_4 & = & 7 \\ & 3x_1 + x_2 & \leq & 24 \\ & x_2 & \geq & 0 \\ & x_3 & \leq & 0 \end{array}$$

Standard form

$$\begin{array}{rrcl} \text{Maximise} & -2x_4 + 2x_5 - 6x_2 + x_6 + x_7 & & \\ \text{Subject to} & x_4 - x_5 - x_6 + x_7 & \leq & 7 \\ & -x_4 + x_5 + x_6 - x_7 & \leq & -7 \\ & 3x_4 - 3x_5 + x_2 & \leq & 24 \\ & x_6 - x_7 & \leq & 0 \\ & x_2, x_4, x_5, x_6, x_7 & \geq & 0 \end{array}$$

Definition 13.5 - *Slack Form*
The simplex algorithm converts standard form linear programs to slack form.
To convert a standard form program to *Slack Form*

i) Set the objective function equal to a new variable z.

ii) For each inequality $\sum_{i\leq j\leq n} a_{ij}x_j \leq b_i$:

  (a) Introduce a new variable x_{n+i}
  (b) Set $x_{n+i} = b_i - \sum_{1\leq j\leq n} a_{ij}x_j$

Example 13.2 - *Converting to Slack Form*
Standard form

$$\begin{array}{rrcl} \text{Maximise} & 2x_1 - 3x_2 + 3x_3 & & \\ \text{Subject to} & x_1 + x_2 - x_3 & \leq & 7 \\ & -x_1 - x_2 + x_3 & \leq & -7 \\ & x_1 - 2x_2 - 2x_3 & \leq & 4 \\ & x_1, x_2, x_3 & \geq & 0 \end{array}$$

Slack form

$$\begin{array}{rcl} z & = & 2x_1 - 3x_2 + 3x_3 \\ x_4 & = & 7 - x_1 - x_2 + x_3 \\ x_5 & = & -7 + x_1 + x + 2 - x_3 \\ x_6 & = & 4 - x_1 + 2x_2 + 2x_3 \end{array}$$

Definition 13.6 - *Basic & Non-Basic Variables*
The *Basic Variables* of a slack form linear program are all the dependent variables, except z.
The *Non-Basic Variables* of a slack form linear program are all the independent variables.

Definition 13.7 - *Basic Solution*
The *Basic Solution* to a slack form linear program is found by setting all the non-basic variables to 0.
If all constants in equations for variables, other than z, are ≥ 0 then this is a feasible solution.

Example 13.3 - *Basic Solution*
Slack Form

$$\begin{aligned} z &= 3x_1 + x_2 + 2x_3 \\ x_4 &= 30 - x_1 - x_2 - 3x_3 \\ x_5 &= 24 - 2x_1 - 2x_2 - 5x_3 \\ x_6 &= 26 - 4x_1 - x_2 - 2x_3 \end{aligned}$$

Basic Solution

$$\begin{aligned} z &= 3.0 + 0 + 0 \\ x_4 &= 30 - 0 - 0 - 0 \\ x_5 &= 24 - 2.0 - 0 - 0 \\ x_6 &= 26 - 0 - 0 - 0 \end{aligned}$$

$\mathbf{x} = (0, 0, 0, 30, 24, 36)$ and $z = 0$.

Definition 13.8 - *Simplex Algorithm Initialisation*
The simplex algorithm assumes the slack form it is given is feasible.
We can ensure this by define a new auxillary linear program, L_{aux} by

1. Changing the objective function to $-x_0$.
2. Subtract $-x_0$ from non-basic part of constraints.
3. Constrain $x_0 \geq 0$.
4. Convert to Slack Form.
5. Perform simplex algoritm on L_{aux}.
   (a) If optimal solution does **not** set x_0 to 0 then the linear program is infeasible.
   (b) Else
      i. If x_0 is basic do a pivot to make it non-basic.
      ii. Change the objective function in the final slack form of L_{aux} to teh original objective function.
      iii. Substitute each basic variable of the objective function with its constraint definition.
      iv. Set x_0 to 0 in all equations
      v. Return

Definition 13.9 - *Simplex Algorithm - Iteration*

i) Select a non-basic variable x_i which has a positive coefficient in the objective function.

ii) Find the maximum value of x_i for each inequality.

iii) Take the basic variable of the inequality which allows for the lowest maximum of x_i.

iv) Rearrange this inequality in terms of x_i.

v) Substitute this rearrangement into the other inequalities and the objective function.

vi) Repeat until the objective function has no non-negative coefficients.

Definition 13.10 - *Pivot*
Pivot is an operation which exchanges the roles of one non-basic variable and one basic variable.

Example 13.4 - *Simple Algorithm - Iteration*
Slack Form

$$\begin{aligned} z &= 3x_1 + x_2 + 2x_3 \\ x_4 &= 30 - x_1 - x_2 - 3x_3 \\ x_5 &= 24 - 2x_1 - 2x_2 - 5x_3 \\ x_6 &= 26 - 4x_1 - x_2 - 2x_3 \end{aligned}$$

Consider x_1, arbitrarily chosen from x_1, x_2 & x_3.
It has max values 30, 12 & 9 for x_4, x_5 & x_6 respectfully.
Thus, we pivot x_1 and x_6.

$$\begin{aligned} z &= 3\left(9 - \frac{x_2}{4} - \frac{x_3}{2} - \frac{x_6}{4}\right) + x_2 + 2x_3 \\ x_4 &= 30 - \left(9 - \frac{x_2}{4} - \frac{x_3}{2} - \frac{x_6}{4}\right) - x_2 - 3x_3 \\ x_5 &= 24 - 2\left(9 - \frac{x_2}{4} - \frac{x_3}{2} - \frac{x_6}{4}\right) - 2x_2 - 5x_3 \\ x_1 &= 9 - \frac{x_2}{4} - \frac{x_3}{2} - \frac{x_6}{4} \end{aligned}$$

Now x_1, x_4, x_5 are the basic variables and the basic solution is $\mathbf{x} = (9, 0, 0, 21, 6, 0),\ z = 27$.
Consider x_3, arbitrarily chosen from x_2 & x_3.
It has max values 8.4, 1.5 & 18 for x_4, x_5 & x_1 respectfully.
Thus, we pivot x_3 and x_5

$$\begin{aligned} z &= \frac{111}{4} + \frac{x_2}{16} - \frac{x_5}{8} - \frac{11x_6}{16} \\ x_4 &= \frac{69}{4} + \frac{3x_2}{16} + \frac{5x_3}{8} + \frac{x_6}{16} \\ x_3 &= \frac{3}{2} - \frac{3x_2}{8} - \frac{x_5}{4} + \frac{x_6}{8} \\ x_1 &= \frac{33}{4} - \frac{x_2}{16} + \frac{x_5}{8} - \frac{5x_6}{16} \end{aligned}$$

Now x_2, x_5, x_6 are the basic variables and the basic solution is $\mathbf{x} = (\frac{33}{4}, 0, \frac{3}{2}, \frac{69}{4}, 0, 0),\ z = \frac{111}{4}$.
Consider x_2, last basic variable with a non-negative coefficient.
It has max values ∞, 4 & 212 for x_4,x_3 & x_1 respectfully.
Thus we pivot, x_2 and x_3.

$$\begin{aligned} z &= 28 - \frac{x_3}{6} - \frac{x_5}{6} - \frac{2x_6}{4} \\ x_4 &= 18 - \frac{x_3}{2} - \frac{x_5}{8} + \frac{x_6}{8} \\ x_2 &= 4 - 8\frac{x_3}{3} - \frac{2x_5}{3} + \frac{x_6}{3} \\ x_1 &= 8 + \frac{x_3}{6} + \frac{x_5}{6} - \frac{x_6}{3} \end{aligned}$$

Now x_3, x_5, x_6 are the basic variables and the basic solution is $\mathbf{x} = (8, 4, 0, 18, 0, 0),\ z = 28$.
All the coefficients in objective function are now negative.
Simplex algorithm terminates.
The original linear program can be maximised by setting $x_1 = 8$ $x_2 = 4$ & $x_3 = 0$.
Thus the maximum objective value is 28.

Definition 13.11 - *Unbounded Linear Program*
An *Unbounded Linear Program* is a linear program that has feasible solutions, but no maximum.

Example 13.5 - *Unbounded Linear Program*
Slack form

$$\begin{aligned} z &= x + y \\ x_1 &= 7 - 2x + 5y \\ x_2 &= 4 - x + 3y \end{aligned}$$

Pivot x_1

$$\begin{aligned} z &= \frac{7}{2} + \frac{7}{2}y - x_1 \\ x &= \frac{7}{2} + \frac{5}{2} - x_1 \\ x_2 &= \frac{1}{2} + \frac{1}{2}y + x_1 \end{aligned}$$

Cannot pivot y as it is unbounded in the equations for x and x_2.

Proposition 13.2 - *Identifying Unbounded Linear Programs*
We can use the simplex algorithm to identify unbounded linear programs.
If, while trying to pivot a variable, you find that the variable is unbounded in all constraint equations then the program is unbounded.

Theorem 13.1 - *Fundamental Theorem of Linear Programming*
Any linear program given in standard form is either

- Has an optimal solution with a finite object value;
- Infeasible; Or,
- Unbounded

Proposition 13.3 - *Run-Time Complexity of Simplex Algorithm*
There have been found examples of data where the simplex algorithm as run-time complexity $\in O(2^n)$.
However, it's average run-time complexity is polynomial, so the simplex algorithm often works well in practice.
N.B. - The *Ellipsoid Algorithm* is an alternative algorithm with polynomial run-time, but is bad in practice.

3.14 Algorithmic Game Theory

Definition 14.1 - *Payoff Matrix*
A *Payoff Matrix* is an $m \times n$ matrix which represents the payoff for each player for all possible strategy combinations of a two person game.
m is the number of possible strategies for row player.
n is the number of possible strategies for column player.
The value in each index is a tuple where the first value is the payoff for the row player, and the second is the payoff for the column player.

Example 14.1 - *Payoff Matrix*
Here is the payoff matrix for a game of rock-paper-scissors

	Rock	Paper	Scissors
Rock	0,0	-1,1	1,-1
Paper	1,-1	0,0	-1,1
Scissors	-1,1	1,-1	0,0

Definition 14.2 - *Payoff Notation*
$R[i,j]$ denotes the payoff for the row player if they choose strategy i and their opponent choses strategy j.
$C[i,j]$ denotes the payoff for the column player if they choose strategy j and their opponent choses strategy i.

Definition 14.3 - *Dominated Strategy*
A strategy i is dominated by a strategy j if you cannot do worse by switching to following j.
This is defined formally as.
Let $m,n\in\mathbb{N}$ be number of row player strategies & number of column player strategies The row player's strategy i is dominated if for some $j\in[1,m]$
$R[i,k]\leq R[j,k]\ \forall\ k\in[1,n]$ and $R[i,k]<R[j,k]$ for some $k\in[1,n]$.
The column player's strategy i is dominated if for some $j\in[1,n]$
$C[k,i]\leq C[k,j]\ \forall\ k\in[1,m]$ and $C[j,i]<C[k,j]$ for some $k\in[1,m]$.

Definition 14.4 - *Strictly Dominated Strategy*
A strategy i is strictly dominated by a strategy j if you always do better by using j instead of i.
This is defined formally as.
Let $m,n\in\mathbb{N}$ be number of row player strategies & number of column player strategies respectively.
The row player's strategy is strictly dominated if for some $j\in[1,m]$
$R[i,k]<R[j,k]\ \forall\ k\in[1,n]$.
The column player's strategy is strictly dominated if for some $j\in[1,n]$
$C[k,i]<C[k,j]\ \forall\ k\in[1,m]$.

Definition 14.5 - *Mixed Strategy*
A *Mixed Strategy* is a strategy where each turn a player plays their i^{th} strategy with probability p_i where $0\leq p_i\leq 1$ and $\sum_i p_i=1$.
N.B. - These are generally represented as row vectors $\mathbf{p}=(p_1,\dots,p_m)$.

Definition 14.6 - *Expected Payoff from Mixed Strategy*
Let S_R & S_C be the set of mixed strategies for the row player & column player, respectively.
Suppose players choose strategies $\mathbf{p}\in S_R$ & $\mathbf{q}\in S_C$.
The expected payoff for row player is

$$R[\mathbf{p},\mathbf{q}]=\sum_{1\leq i\leq m}\sum_{1\leq i\leq n}p_iq_jR[i,j]$$

The expected payoff for column player is

$$C[\mathbf{p},\mathbf{q}]=\sum_{1\leq i\leq m}\sum_{1\leq i\leq n}p_iq_jC[i,j]$$

Definition 14.7 - *Nash Equilibrium*
A *Nash Equilibrium* is a pair of mixed strategies $(\mathbf{p},\mathbf{q})$, for the row & column players respectively, where neither player can increase their payoff by changing their strategy and their opponent not changing.
This is defined formally as.
A Nash Equilibrium for a 2 Player game is a pair $(\mathbf{p},\mathbf{q})$ $\mathbf{p}\in S_R,\ \mathbf{q}\in S_C$, which satisfies

i) $R[\mathbf{p},\mathbf{q}]\geq R[\mathbf{p}',\mathbf{q}]\ \forall\mathbf{p}'\in S_R$;

ii) And, $C[\mathbf{p},\mathbf{q}]\geq C[\mathbf{p},\mathbf{q}']\ \forall\mathbf{q}'\in S_C$.

Proposition 14.1 - *Proving a Nash Equilibrium*
We know the expected payoffs

$$R[p',q]=\sum_{i=1}^n p'_iR[i,q] \ \& \ R[p,q']=\sum_{i=1}^n p'_iC[[p,i]$$

Thus to prove that (p,q) is a *Nash Equilibrium* it is enough to check that

$$R[p,q]\geq R[i,q]\ \forall i\in[1,m] \ \& \ C[p,q]\geq C[i,q]\ \forall i\in[1,n]$$

Example 14.2 - *Proving a Nash Equilibrium*
Consider the following payoff matrix

	Curry	Pizza
Curry	(1,5)	(0,0)
Pizza	(0,0)	(5,1)

Let $p=(\frac{1}{6},\frac{5}{6})$ & $q=(\frac{5}{6},\frac{1}{6})$ be mixed strategies chosen by the row and column player respectively.
We shall show that (p,q) is a *Nash Equilibrium.*
We need to check that $R[p,q]\geq R[1,q],\ R[p,q]\geq R[2,q],\ C[p,q]\geq C[p,1]$ & $C[p,q]\geq R[q,2]$.
$R[p,q]=\frac{1}{6}.\frac{5}{6}.1+\frac{1}{6}.\frac{1}{6}.0+\frac{5}{6}.\frac{5}{6}+\frac{5}{6}.\frac{1}{6}.5=\frac{5}{6}$.
$C[p,q]=\frac{1}{6}.\frac{5}{6}.1+\frac{1}{6}.\frac{1}{6}.0+\frac{5}{6}.\frac{5}{6}.0+\frac{5}{6}.\frac{1}{6}.1=\frac{5}{6}$.
$R[1,q]=1.\frac{5}{6}.1+0+0+0=\frac{5}{6}\leq R[p,q]$.
$R[2,q]=0+0+0+1.\frac{1}{6}.5=\frac{5}{6}\leq R[p,q]$.
$C[p,1]=\frac{1}{6}.1.5+0+0+0=\frac{5}{6}\leq R[p,q]$.
$C[p,2]=0+0+0+\frac{5}{6}.1.1=\frac{5}{6}\leq R[p,q]$.
Thus, (p,q) is a *Nash Equilibrium.*

Theorem 14.1 - *Possibility of Nash Equilibrium*
Every game that can be specified by a payoff matrix has at least one *Nash Equilibrium.*

Proposition 14.2 - *Consequences of Nash Equilibrium*
After playing a Nash Equilibrium you can say *given what the other player did, I have no regrets.*
And if you believe the other player will play their part of a Nash Equilibrium, then you cannot get a better payoff unless you play your part - So you play you part.

Definition 14.8 - *Zero-Sum Game*
A *Zero-Sum Game* is a game where $R[i,j]+C[i,j]=0\ \forall\ i\in S_R,\ j\in S_C$.

3.15 Max-Flow Problem

Definition 15.1 - *Max-Flow Problem*
Consider have being given a flow network[1].
The *Max-Flow Problem* is to find the maximum flow value for this network.

[1]See **Subsection 2.8**

Definition 15.2 - *Ford-Fulkerson Method*
The *Ford-Fulkerson Method* solves the max-flow problem.
The *Ford-Fulkerson Method* can be performed on the flow network G with source s & sink t using the following pseudocode

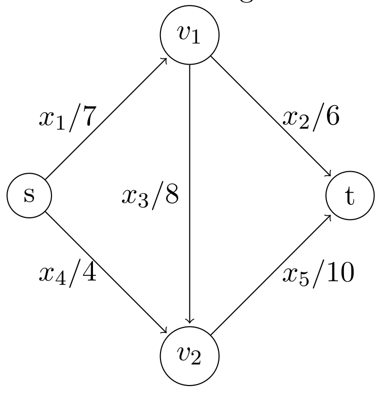
```
FORD-FULKERSON(G, s, t)
for ((u,v)∈G):
  f(u,v)=0
  while (∃ augmenting path $p$ in $G_f$):
    $c_f$(p)=min{$c_f$(u,v):(u,v)∈p)
    for ((u,v)∈p):
      f(u,v)=f(u,v)+$c_f$(p)
      f(v,u)=f(v,u)−$c_f$(p)
return f
```

Theorem 15.1 - *Run-Time Complexity of Ford-Fulkerson Method*
The *Run-Time Complexity* of the Ford-Fulkerson Method depends on the order that the augmentation paths are chosen.
Each iteration has run-time complexity $\in O(|E|)$.
The flow value increases by at least one unit with each iteration.
The number of iterations is at most $|f*|$ times.
The total *Run-Time Complexity* for the Ford-Fulkerson Method $\in O(|f*||E|)$.
N.B. - $|f*|$ is the maximum flow value.

Definition 15.3 - *Edmonds-Karp Algorithm*
The *Edmonds-Karp Algorithm* is a solution to the max-flow problem.
The *Edmonds-Karp Algorithm* is an adjustment to the Ford-Fulkerson Method such that it's run-time complexity does not depend upon the maximum flow value.
At each iteration the *Edmonds-Karp Algorithm* chooses the path from $s \to t$ with the minimum number of edges as the augmenting path.

Theorem 15.2 - *Run-Time Complexity of Edmonds-Karp Algorithm*
The *Edmonds-Karp Algorithm* has run-time complexity $\in O(|V||E|^2)$.

Example 15.1 - *Every Max-Flow Problem is a Linear program*
Given the following flow network



The max-flow problem for this flow network can be described by the following linear program

$$\begin{array}{rrcl}
\textit{Maximise} & x_1 + x_4 & & \\
\textit{Conditions} & x_1 & \leq & 7 \\
& x_2 & \leq & 6 \\
& x_3 & \leq & 8 \\
& x_4 & \leq & 4 \\
& x_5 & \leq & 10 \\
& x_1 - x_2 - x_3 & = & 0 \\
& x_3 + x_4 - x_5 & = & 0 \\
& x_1, x_2, x_3, x_4, x_5 & \geq & 0
\end{array}$$

4 Reference

4.1 Algorithmic Strategy

Definition 1.1 - *Decision Problems*
A *Decision Problem* is a problem that has a yes or no answer.

Definition 1.2 - *Iteration*
Iteration is an algorithmic strategy for analysing n pieces of data.
The general process is

i) Process inputs $x_1, \dots, x_{n-1}$.

ii) Combine this output with input x_n.

Proposition 1.1 - *Time Complexity of Iteration Strategy*
Typically the time complexity for the *worst-case scenario* of an iteration startegy recurrence is given by

$$T(n) = aT(n-1) + f(n)$$

Definition 1.3 - *Divide-&-Conquer*
Divide-&-Conquer is an algorithmic strategy for analysing n pieces of data.
The general process is

i) Divide inputs into a fixed number of groups.

ii) Process a result for each group.

iii) Combine the results for a final output.

Proposition 1.2 - *Time Complexity of Divide-&-Conquer Strategy*
The typical time complexity for the *worst-case scenario* of a Divide-&-Conquer recurrence is given by

$$T(n) = aT\left(\left\lfloor \frac{n}{a} \right\rfloor\right) + f(n)$$

Remark 1.1 - *Iteration vs Divide-&-Conquer*
By comparing the general time complexity formulae for the *worst-case scenarios* of *divide-&-conquer* to *iteration* we see $T(n) \in \Theta(a^n)$ for iteration and $T(n) \in \Theta(n \log_2 n)$ for divide-&-conquer.
This shows us that divide-&-conquer algorithms tend to have better *worst-case* asymptotic behaviour than iterative algorithms.
Definition 1.4 - *In Place Algorithm*
An *In Place Algorithm* does not require any additional data structures to complete their process.
E.G. - Merge Sort.

Definition 1.5 - *Out of Place Algorithm*
An *Out of Place Algorithm* require additional data structures to complete their process.
E.G. - Heap Sort.

4.2 Regular Expression

Definition 2.1 - *Regular Expression, REGEX*
Regular Expression is a format language used to produce patterns for strings which are not exact strings, but fulfil given criteria.
Here are common features of *REGEX*

- $a+$, one or more as.
- $a*$, zero or more as.
- $a?$, one or zero as.
- an, exactly n as.
- $(a|b)$, a or b.
- $[0-9]$, any digit 1 to 9.
- $[a-m]$, any character a to m.
- . , any character

4.3 Mathematics

Definition 3.1 - *Degree of a Polynomial*
The *Degree* of a polynomial, $f(x)$, is the greatest power of x with a non-zero coefficient.

Definition 3.2 - *Degree-Bound*
An integer is a *Degree-Bound* of a given polynomial if it is greater in value than the degree of the polynomial.

Definition 3.3 - *Coefficients Representation of Polynomials*
Let $A(x)$ be a polynomial of degree $n-1$.
Then A can be represented by an array of n numbers where the i^{th} element is the coefficient of x_i.

$$A = a_0 + a_1x + a_2x^2 + \cdots + a_{n-1}x^{n-1} = (a_0, a_1, a_2, \dots, a_{n-1})$$

Proposition 3.1 - *Addition of Polynomials in Coefficient Representation*
Let $A(x)$ & $B(x)$ be two polynomials in coefficient representation so $A = (a_0, a_1, \dots, a_{n-1})$ & $B = (b_0, b_1, \dots, b_{n-1})$.
Then

$$A + B = (a_0 + b_0, a_1 + b_1, a_2 + b_2, \dots, a_{n-1} + b_{n-1})$$

N.B. - The runtime of this is $\in O(n)$.

Remark 3.1 - *Point-Value Representation of Polynomials*
A polynomial $A : \mathbb{R}^a \to \mathbb{R}^b$ of degree-bound n can be uniquely represented by n points in $\mathbb{R}^b$.

$$A(x) = \{(x_0, y_0), (x_1, y_1), \dots, (x_{n-1}, y_{n-1})\}$$

Proposition 3.2 - *Addition of Point-Value Represented Polynomials*
Let $A(x)$ & $B(x)$ be point-value represented polynomials where $A = \{(x_0, a_0), (x_1, a_1), \dots, (x_{n-1}, a_{n-1})\}$ & $B = \{(x_0, b_0), (x_1, b_1), \dots, (x_{n-1}, b_{n-1})\}$.

$$A + B = \{(x_0, a_0 + b_0), (x_1, a_1 + b_1), \dots, (x_{n-1}, a_{n-1} + b_{n-1})\}$$

N.B. - The runtime of this is $\in O(n)$.

Proposition 3.3 - *Multiplication of Point-Value Represented Polynomials*
Let $A(x)$ & $B(x)$ be point-value represented polynomials where $A = \{(x_0, a_0), (x_1, a_1), \dots, (x_{n-1}, a_{n-1})\}$ & $B = \{(x_0, b_0), (x_1, b_1), \dots, (x_{n-1}, b_{n-1})\}$.

$$A \times B = \{(x_0, a_0.b_0), (x_1, a_1.b_1), \dots, (x_{n-1}, a_{n-1}.b_{n-1})\}$$

N.B. - The runtime of this is $\in O(n)$.

Definition 3.4 - *Complex Roots of Unity*
Complex Roots of Unity of degree $n \in \mathbb{N}$ are given by the equation

$$\omega_n^j = e^{\frac{i2\pi j}{n}} \quad j \in (0, n]$$

Theorem 3.1 - *Cancellation Rule for Complex Roots of Unity*
For $d \in \mathbb{R}$ and $n, k \in \mathbb{N}$

$$\omega_{dn}^{dk} = \omega_n^k$$

Theorem 3.2 - *Halving Rule for Complex Roots of Unity*
If $n > 0$ is even, then the squares of the n complex roots of unity are the same as those of the $\frac{n}{2}$ roots of unity.

$$(\omega_N^k) = \omega_N^{2k} = \omega_{N/2}^k$$

N.B. - This is an interpretation of the *Cancellation Rule.*

Theorem 3.3 - *Euler's Formula*
Euler's Formula describes the relationship between trigonometry and complex exponential functions.
For $x \in \mathbb{R}$

$$e^{ix} = \cos x + i \sin x$$

Definition 3.5 - *Linear Function*
A *Linear Function* has one independent variable and one dependent variable, and form a straight line on a two dimensional space.
E.g. - $y = ax + b$.

Definition 3.6 - *Linear Inequality*
A *Linear Inequality* is an inequality that involves a linear funciton.

Definition 3.7 - *Half-Space*
A *Half-Space* is the area defined by a linear inequality in two dimensional space.

Definition 3.8 - *Incompatible Linear Inequalities*
A set of linear inequalities are said to be *incompatible* if the half spaces they define don't overlap.

Definition 3.9 - *Feasible Region*
A *Feasible Region* is the region defined by multiple linear inequalities.
Feasible regions may be bounded or unbounded.