Extended Algorithms and Programming Techniques
COMP3821 UNSW

Jeremy Le
(Based of Hussain Nawaz's COMP3121 Notes)

2023T1

Contents

1 Mathematical Reminders

1.1 Geometric Series

Claim:

$$\text{If } r \neq 1 \text{ then } \sum_{k=0}^{m} r^k = \frac{r^{m+1}-1}{r-1}$$

Proof.

$$\begin{aligned}
(r-1)\sum_{k=0}^{m} r^k &= (r-1)(r^m + r^{m-1} + \cdots + r + 1) \\
&= (r^{m+1} + r^m + \cdots + r^2 + r) - (r^m + r^{m-1} + \cdots + r + 1) \\
&= r^{m+1} - 1
\end{aligned}$$

1.2 Basic logarithm identity

Claim:

$$\text{If } a, b, c > 0 \text{ then } a^{\log_b c} = c^{\log_b a}$$

Proof.

$$\begin{aligned}
\log_b c \cdot \log_b a &= \log_b a \cdot \log_b c && \text{(because } \times \text{ is commutative)} \\
\log_b(a^{\log_b c}) &= \log_b(c^{log_b a}) && \text{(because} y \log_b x = log_b x^y) \\
a^{\log_b c} &= c^{\log_b a} && (\log_b \text{is injective:} \log_b y = log_b x \implies y = x)
\end{aligned}$$

1.3 Asymptotic notations

Big O Notation We say $f(n) = O(g(n))$ if there exists a positive constants c, N such that

$$0 \leq f(n) \leq cg(n) \quad \forall n \geq N.$$

We may refer to $g(n)$ to be the asymptotic upper bound for $f(n)$.

Big Omega Notation We say $f(n) = \Omega(g(n))$ if there exists positive constants c, N such that

$$0 \leq cg(n) \leq f(n) \quad \forall n \geq N.$$

Then, $g(n)$ is said to be an asymptotic lower bound for $f(n)$. It is useful to say that a problem is at least $\Omega(g(n))$.

Big Theta Notation We say $f(n) = \Theta(g(n))$ if and only if

$$f(n) = O(g(n)) \text{ and } f(n) = \Omega(g(n)).$$

That is, both f and g have the same asymptotic growth.

2 Stable Matchings and the Gale-Shapely Algorithm

2.1 Stable Matching Problem

Setting: Assume that you are running a speed dating agency and have n men and n women as customers. They all attend a dinner party; after the party

- every man gives you his ranking of all the women present, **and**
- every woman gives you her ranking of all men present;

Task: Design an algorithm which produces a *stable matching*: a set of n pairs $p = (m, w)$ of a man m and a woman w so that the following the situation never happens:

for two pairs $p = (m, w)$ and $p' = (m', w')$:

- man m prefers woman w' to woman w, **and**
- woman w' prefers man m to man m'.

Existence A stable matching exists for every possible collection of n lists of preferences provided by all men, and n lists of preferences provided by all women.

Brute Force Takes $n! \approx (n/e)^n$ time to form n couples.

2.2 Gale-Shapley Algorithm

Assumptions

- Produces pairs in stages, with possible revisions;
- A man who has not been paired with a woman will be called *free*.
- Men will be proposing to women.
- Women will decide if they accept a proposal or not.

Algorithm Start with all men free;

While there exists a free man who has not proposed to all women pick such a free man m and have him propose to the highest-ranking woman w on his list to whom he has not proposed yet;

If no one has proposed to w yet she always accepts and a pair $p = (m, w)$ is formed;
Else she is already in a pair $p' = (m', w)$;
  If m is higher on her preference list than m' the pair $p' = (m', w)$ is deleted;
  m' becomes a free man;
  Else m is lower on her preference list than m';
  the proposal is rejected and m remains free.

Proving termination after n^2

- In every round of the *While* loop one man proposes to one woman;
- every man can propose to a woman at most once;
- thus, every man can make at most n proposals;
- there are n men, so in total they can make $\leq n^2$ proposals

Thus the *While* loops can be executed no more than n^2 many times.
With appropriate data structures, the Gale-Shapley alg. runs in $O(n^2)$.

Proving Production of Matching Proof (by contradiction).

- Assume that the *While* loop has terminated, but m is still free.
- This means that m has already proposed to every woman.
- Thus, every woman is paired with a man, because a woman is not paired with anyone only if no one has made a proposal to her.
- But this would mean that n women are paired with all of n men so m cannot be free.

Contradiction!

Proving Stable Matching Proof (by contradiction). Note that during the *While* loop:

- a woman is paired with men of increasing ranks on her list;
- a man is paired with women of decreasing ranks on his list.

Assume now the opposite, that the matching is not stable;
Thus, there are two pairs $p = (m, w)$ and $p' = (m', n')$ such that:

$$m \text{ prefers } w' \text{ over } w; \qquad\qquad w' \text{ prefers } m \text{ over } m'.$$

- m prefers w' over w, so m has proposed to w' before proposing to w;
- Since he is paired with w, woman w' must have either:
  - rejected him because she was already with someone she prefers, or
  - dropped him later after a proposal from someone she prefers;
- In both cases she would now be with m' whom she prefers over m.

Contradiction!

3 Divide and Conquer

3.1 Foundations for Divide and Conquer

Method

- Split the data into 2 or more parts (Divide)
- Solve the corresponding sub-problems by recursion (Conquer)
- Combine the solutions of the sub-problems into a solution.

Complexity (runtime) Assume:

- n is the input size
- we divide in a parts
- each part has size $\frac{n}{b}$
- Combining solutions costs $f(n)$

Then the runtime T of such an algorithm satisfies the equation

$$T(n) = aT\left(\frac{n}{b}\right) + f(n).$$

3.2 Integer Addition and Multiplication

Notation and Basic Methodology We let n be the number of bits in the integer. Addition occurs by moving from the least to most significant bit, adding each bit at a time in $O(n)$. Multiplication is much of the same but, in $O(n^2)$.

The Karatsuba Trick This happens in $O(n^{\log_2 3})$.

3.3 Matrix multiplication

Brute Force Computation The product of multiplying two $n \times n$ matrices is a matrix of size $n \times n$, so n^2 entries. For each entry in that product we do n multiplications. So matrix product by brute force is $\Theta(n^3)$.

Strassen's Algorithm This happens in $\Theta(n^{\log_2 7})$.

3.4 Master Theorem

Setup Master Theorem Let $a \geq 1$ be an integer and $b > 1$ be a real number, $f(n) > 0$ be a non-decreasing function defined on the positive integers. Then, $T(n)$ is the solution of the recurrence

$$T(n) = aT\left(\frac{n}{b}\right) + f(n).$$

Master Theorem

1. If $f(n) = O(n^{\log_b a - \epsilon})$ for some $\epsilon > 0$ then, $T(n) = \Theta(n^{\log_b a})$.
2. If $f(n) = \Theta(n^{\log_b a})$ then, $T(n) = \Theta(n^{\log_b a} \log n)$.
3. If $f(n) = \Omega(n^{\log_b a + \epsilon})$ for some $\epsilon > 0$ and, for some $c < 1$, and some n_0,

$$af(\frac{n}{b}) \leq cf(n)$$

holds for all $n > n_0$ then, $T(n) = \Theta(f(n))$.

If the conditions above do not hold then, the master theorem is not applicable.

3.5 Polynomial Interpolation

From Coefficient to Value Representation Every polynomial $A(x)$ of degree d is uniquely determined by its values at any $d+1$ distinct input values $x_0, x_1, \ldots, x_d$:

$$A(x) \leftrightarrow \{(x_0, A(x_0)), (x_1, A(x_1)), \ldots, (x_d, A(x_d))\}$$

For $A(x) = \boldsymbol{a}_d x^d + \boldsymbol{a}_{d-1} x^{d-1} + \cdots + \boldsymbol{a}_0$, these values can be obtained via a matrix multiplication:

$$\begin{pmatrix} 1 & x_0 & x_0^2 & \ldots & x_0^d \\ 1 & x_1 & x_1^2 & \ldots & x_1^d \\ \vdots & \vdots & \vdots & \vdots & \vdots \\ 1 & x_d & x_d^2 & \ldots & x_d^d \end{pmatrix} \begin{pmatrix} \boldsymbol{a}_0 \\ \boldsymbol{a}_1 \\ \vdots \\ \boldsymbol{a}_d \end{pmatrix} = \begin{pmatrix} A(x_0) \\ A(x_1) \\ \vdots \\ A(x_d) \end{pmatrix}$$

Such a matrix is called the *Vandermonde matrix*.

From Value to Coefficient Representation It can be shown that if x_i are all distinct then this matrix is invertible. Thus if, all x_i are all distinct, given any values $A(x_0), A(x_1), \ldots, A(x_d)$ the coefficients $\boldsymbol{a}_0, \boldsymbol{a}_1, \ldots, \boldsymbol{a}_d$ of the polynomial $A(x)$ are uniquely determined:

$$\begin{pmatrix} \boldsymbol{a}_0 \\ \boldsymbol{a}_1 \\ \vdots \\ \boldsymbol{a}_d \end{pmatrix} = \begin{pmatrix} 1 & x_0 & x_0^2 & \ldots & x_0^d \\ 1 & x_1 & x_1^2 & \ldots & x_1^d \\ \vdots & \vdots & \vdots & \vdots & \vdots \\ 1 & x_d & x_d^2 & \ldots & x_d^d \end{pmatrix}^{-1} \begin{pmatrix} A(x_0) \\ A(x_1) \\ \vdots \\ A(x_d) \end{pmatrix}$$

3.6 Counting Inversions

Brute Force An easy way to count the total number of inversions between two lists is by looking at all pairs $i < j$ of items on one list and determining if they are inverted in the second list, but this would produce a quadratic time algorithm, $T(n) = \Theta(n^2)$.

Divide and Conquer Method The main idea is to tweak the Merge Sort algorithm, by extending it to recursively both sort an array A and determine the number of inversions in A. This can be done much more efficiently, in time $\Theta(n \log n)$.

We split the array A into two equal parts A_{top} and A_{bottom}. We may sort both A_{top} and A_{bottom}. Then, we seek to merge the arrays together. Every time we pull an element from A_{bottom}, such an element is in an inversion with all the remaining elements in A_{top} and we add the total number of elements remaining in A_{top} to the total number of inversions.

3.7 Discrete Fourier Transform

For $\boldsymbol{a} = \langle a_0, a_1, \ldots, a_{n-1}\rangle$ a sequence of n real or complex numbers. We can form the corresponding polynomial $P_A(x) = \sum_{j=0}^{n-1} a_j x^j$, and evaluate it at all complex roots of unity of order n:

$$\text{For all } 0 \leq k \leq n-1, \text{ we compute } P_A(w_n^k) = A_k = \sum_{j=0}^{n-1} a_j w_n^{jk}.$$

The DFT of a sequence a is a sequence A of the same length.

Inverse Discrete Fourier Transform The **IDFT** of a sequence $\boldsymbol{A} = \langle A_0, A_1, \ldots, A_{n-1}\rangle$ is the sequence of values $\boldsymbol{a} = \langle a_0, a_1, \ldots, a_{n-1}\rangle = \langle \frac{P_a(1)}{n}, \frac{P_a(\omega_n^{-1})}{n}, \ldots, \frac{P_a(\omega_n^{1-n})}{n} \rangle$.
We can show that IDFT(DFT(a) $= a$ and DFT(IDFT(A)) $= A$.

Computation Brute force computation of the DFT takes $\Theta(n^2)$, same for IDFT. The DFT of a sequence can be computed in $\Theta(n \lg n)$ using the FFT (as can be the IDFT).

3.8 Convolution

(Linear) Convolution

$$A \star B = \langle \boldsymbol{c}_0, \boldsymbol{c}_1, \ldots, \boldsymbol{c}_{n+m}\rangle \text{ where } \boldsymbol{c}_j = \sum_{i+k=j} \boldsymbol{a}_i \boldsymbol{b}_k.$$

Interpretation in terms of Polynomials Form the two corresponding polynomials and multiply them $C(x) = A(x) \cdot B(x)$

$$A(x) = \sum_{i=0}^{n} \boldsymbol{a}_i x^i \qquad\qquad B(x) = \sum_{k=0}^{m} \boldsymbol{b}_k x^k$$

$$C(x) = \sum_{j=0}^{m+n} \left(\sum_{i+k=j} \boldsymbol{a}_i b_k \right) x^j = \sum_{j=0}^{n+m} \boldsymbol{c}_j x^j$$

The sequence of coefficients of the product polynomial is the convolution of the coefficients of the factors: $\langle \boldsymbol{c}_0, \boldsymbol{c}_1, \ldots, \boldsymbol{c}_{n+m}\rangle = \langle \boldsymbol{a}_0, \boldsymbol{a}_1, \ldots, \boldsymbol{a}_n\rangle \star \langle \boldsymbol{b}_0, \boldsymbol{b}_1, \ldots, \boldsymbol{b}_m\rangle$.

For a more visual understanding watch: 3Blue1Brown's video on convolutions here.

4 Greedy Algorithms

4.1 Foundations for The Greedy Method

Method Search for an admissible solution of maximal reward (/minimal cost)

- Introduce problem *elements*
- Establish which combinations of elements are *admissible*
- Define a *quality* measure on problem's elements
- Build a solution step by step by adding elements of highest quality

Optimality proof method (exchange argument)

- Pick any solution S
- morph it by swapping elements with higher quality ones
- show that any swap leads to a solution with higher reward
- stop when we arrive at the greedy solution G
- Conclude that the reward G is larger than S

4.2 Activity Selection Problem

Setting A list of activites $a_i, (1 \leq i \leq n)$ with starting times s_i and finishing times f_i. No two activities can take place simultaneously.

Task Find a *maximum size* subset of compatible activities.

Solution Among the activities which do not conflict with the previously chosen activities always chose the one with the earliest end time.

Proof Claim: any solution S has $\leq$ number of activities than the greedy solution G.

1. Find the first place where the chosen activity violates the greedy choice.
2. Show that replacing that activity with the greedy choice produces a non-conflicting selection S' with the same number of activities.
3. Continue until all activities match those in the greedy solution G.

Complexity We sort activities using their finishing times in increasing order in $O(n \log n)$ time. Then loop through all activities linearly for a total time of $O(n \log n)$.

Setting A list of activities $a_i, (1 \leq i \leq n)$ with starting times s_i and finishing times $f_i = s_i + d$. Thus, all activities are of the same duration. No two activities can take place simultaneously.

Task Find a subset of compatible activities of *maximal total duration*.

Solution Since all activities are of the same duration, this is equivalent to finding a selection with the largest number of non-conflicting activities, i.e., the previous problem.

A greedy strategy no longer works - we need a more sophisticated technique.

4.3 Dijkstra's Shortest Path Algorithm

Updating our Heap Data Structure We will use heaps represented by arrays; the left child of $A[i]$ is stored in $A[2i]$ and the right child in $A[2i+1]$. We will store in heaps vertices of graphs with key computed in various ways; if a graph has n vertices we will label them with positive integers 1 to n. Thus every element of A is of the form $(i, k(i))$ where $k(i)$ is the key of element i.

Besides the array A which represents the heap, we will use another array P of the same length which stores the position of elements in the heap; thus $A[P[i]] = (i, k(i))$. Changing the key of an element i is now an $O(\lg n)$ operation: we look up its position $P[i]$ in A, change the key of the element in $A[P[i]]$ and then perform the Heappify operation to make sure the Heap property is being preserved.

Setting Let $G = (V, E)$ be a directed graph with non-negative weight $w(e) \geq 0$ assigned to each edge $e \in E$. We are also given a vertex $v \in V$. For simplicity, we assume that any $u \in V$ can be reached from v.

Task Find for every $u \in V$ the shortest path from v to u.

Algorithm Starting from a set of vertices $S = \{v\}$ which contains a single source vertex. At each stage of construction we add the element $u \in V \setminus S$ which has the shortest path from v to u with all intermediate vertices already in S.

Correctness Assume that there exists a shorter path from v to u in G. By our choice of u such a path cannot be entirely in S. Let z be the first vertex outside S on such a shortest path. But then the path from v to such z would be shorter than the path from v to u, contradicting our choice of u.

Efficient Implementation

1. All vertices expect v placed in heap with additional position array, weights $w(u, v)$ if $(v, u) \in E$ or ∞ as the key.
2. Key of each element u will be updated with length $lh_{S,v}(u)$ of the shortest path from v to u which has all intermediate vertices on such a path in S.

3. Pop the element u from the priority queue with smallest key and add to S.

4. For all elements $z \in V \setminus S$ for which $(u, z) \in E$, if $lh_{S,v}(u) + w(u, z) < lh_{s,v}(z)$ update key of z to $lh_{S,v}(u) + w(u, z)$.

Complexity For a graph with n vertices and m edges, each edge is inspected only once, and popping an element with the smallest key and updating a vertex key takes $O(\lg n)$ many steps each. So in total, the algorithm runs in $O(m \lg n)$ time.

4.4 Huffman Code

Encoding Texts Given a set of symbols you want to encode these symbols using binary strings, so that sequences of such symbols can be decoded in an unambiguous way.

Fixed-width Encodings Reverse bit strings of equal and sufficient length, given the number of distinct symbols to be encoded. This is the main idea behind the ASCII code.

Towards Variable-Width Encoding The previous method is not economical: all symbols have codes of equal length. One would prefer an encoding in which frequent symbols have short codes while infrequent ones can have longer codes.

Prefix Code The previous method was unable to partition a bitstream uniquely into segments. To do this we use a prefix code. A prefix code is a map from symbols to bit sequences such that no code of a symbol is a prefix of a code for another symbol.

The Huffman Code Given the frequencies of each symbol, design an optimal prefix code, i.e. a prefix code such that the expect length of an encoded text is as small as possible.

4.5 Union-Find

Three Operations

- `MakeUnionFind(S)` - Given a set S returns a structure in which all elements are placed into distinct singleton sets. Runs in $O(n)$ time where $n = |S|$.

- `Find(v)` - Given a vertex v, returns the set to which v belong. Runs in $O(1)$ time.

- `Union(A, B)` - Given two sets A, B, changes the data structure by replacing sets A and B with the set $A \cup B$. Initial sequence of k consecutive `Union` operations runs in time $O(k \lg k)$.

  – Run time of single `Union` not given. This approach is amortized analysis.

  – *inital* sequence of k `Union` operations means we start with all sets being singletons and then apply k `Union` operations.

  – *consecutive* sequence of `Union` means a sequence of `Union` operations possibly interspered with some `FIND` operations but not other `Union` operations not belonging to the considered sequence of k `Union` operations.

Implementation The simplest implementation of the UF data structure consists of:

1. an array A such that $A[i] = j$ means that i belongs to set labeled by j;
2. an array B such that $B[i]$ contains the number of elements in the set i and pointers to the first and last elements of the list of elements in the set i.

`Union`(i, j) if defined as follows: if the set labeled by i has $\geq$ elements than the set labeled by j then labels in array A of all elements in the set labeled by j is changed to i and array B is updated accordingly. Else do the opposite.

Complexity Any sequence of k initial consecutive `Union` operations can touch at most $2k$ elements of S. Every `Union` operation at least doubles the size of the set and could change fewer than $\lg 2k$ many times. Thus any sequence of k initial consecutive `Union` operations will have in total fewer than $2k \lg 2k$ many label changes. Thus, every sequence of k inital consecutive `Union` operations has time complexity of $O(k \lg k)$.

4.6 Minimum Spanning Trees

Let $G = (V, E)$ be a connected undirected graph.

Spanning Tree A *spanning tree* is a subgraph $T = (V, E_T)$ of G such that T does not contain any cycle and is connected.

Minimum Spanning Tree If G is a (edge-) weighted graph, then a *minimum spanning tree* is a spanning tree of minimum weight.

Kruskal's Algorithm Sort all edges E in non-decreasing order by weight. Then, starting from the lowest weight to highest, if adding an edge will not result in a cycle, then add it to the graph. Otherwise, discard that edge. The process terminates when the list of all edges has been exhausted.

Correctness (Spanning Tree) Let T be the output of the algorithm, we know that T does not contian any cycle. Assume there are two or more connected components C_1 and C_2. G is connected, so there are some edges connecting C_1 to C_2 in G. The first of such edges would have been added to T because it would not create any cycle in T. So T is a spanning tree.

(Minimality) We consider the case where all weights are distinct. Let T be the output of KA. Consider a spanning tree T' distinct from T. Let $e = \{u, v\}$ be the smallest-weight edge in T that is not in T'. T' is spanning so there exists a path P from u to v. T has no cycles, so there exists an edge $f \in P$ that is not in T. Let $T'' = (V, \{e\} \cup E_{T'} \setminus \{f\})$; it is a spanning tree. $w(e) < w(f)$ because otherwise KA would have added f to T instead of e. Furthermore, T'' weighs less than T', so T' is no an MST. G has an MST and any $T' \neq T$ is not an MST, so T is an MST.

Implementation The `Union-Find` data structure lets us efficiently implement Kurskal's algorithm on graph $G = (V, E)$ with n vertices and m edges. We first sort m edges which takes $O(m \lg m)$. Since $m \leq n^2$ this step also takes $O(m \lg n^2) = O(m \log n)$. For the algorithm we making connected components and merge them into a single connected component which is the same as `Union-Find`.

For each edge $e = (v, u)$ we use two `Find` operations `Find(`u`)` and `Find(`v`)` to determine if they belong in the same component. If they are we add edge $e = (u, v)$ to the spanning tree and perform `Union(`i, j`)` to place u and v into the same connected component.

In total we perform $2m$ `Find` operations, each costing $O(1)$, in total coasting $O(m)$. We also perform $n - 1$ `Union` operations which cost $O(n \lg n)$. In total, together with the initial sorting the time complexity is $O(m \log n)$.

5 Maximum Flow

5.1 Flow Networks

Flow Network A *flow network* $G = (V, E)$ is a directed graph where each edge $e = (u, v) \in E$ has a positive integer capacity $c(u, v) > 0$.

There are two distinguished vertices. A source s and sink t. There are no outgoing edges for a sink and likewise, no incoming edges for a source.

Flow A *flow* in G is a function $f : E \to \mathbb{R}^+$, $f(u, v) \geq 0$. which satisfies

1. **Capacity Constraints:** for all edges $e(u, v) \in E$ we require $f(u, v) \leq c(u, v)$.
2. **Flow Conservation:** For all $v \in V \setminus \{(s, t)\}$, we require

$$\sum_{(u,v) \in E} f(u, v) = \sum_{(v,w) \in E} f(v, w).$$

   That is, the incoming flow must be equal to the outgoing flow.

Value of flow The *value of the flow* is defined as

$$|f| = \sum_{v:(s,v) \in E} f(s, v) = \sum_{v:(v,t) \in E} f(v, t).$$

Residual Flow Network The *residual flow network* for a flow network with some flow in it: the network with the leftover capacities.

Augmenting Path Residual flow networks can be used to increase the total flow through the network by adding an *augmenting path*.

The capacity of an augmenting path is the capacity of its "bottleneck" edge, i.e., the capacity of the smallest capacity edge on that path.

We should then send that amount of flow along the augmenting path, recalculating the flow and the residual capacities for each edge used.

5.2 Ford-Fulkerson Algorithm

Ford-Fulkerson algorithm for finding maximal flow in a flow network:

- Keep adding flow through new augmenting paths for as long as it is possible.
- When there are no more augmenting paths, you have achieved the largest possible flow in the network.

Cut A *cut* in a flow network is any partition of the vertices of the underlying graph into two subsets S and T such that:

1. $S \cup T = V$
2. $S \cap T = \emptyset$
3. $s \in S$ and $t \in T$.

Capacity of a Cut The *capacity* $c(S,T)$ *of a cut* (S,T) is the sum of capacities of all edges leaving S and entering T, i.e.

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

Note that the capacities of edges going in the opposite direction, i.e., from T to S do not count.

Flow of Cut The *flow through a cut* $f(S,T)$ is the total flow through edges from $StoT$ minus the total flow through edges from T to S:

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

Clearly, $f(S,T) \leq c(S,T)$ because for every edge $(u,v) \in E$ we assumed $f(u,v) \leq c(u,v)$ and $f(u,v) \geq 0$.

Max Flow Min Cut Theorem The maximal amount of flow in a flow network is equal to the capacity of the cut of minimal capacity.

Edmonds-Karp Algorithm The Edmonds-Karp algorithm improves the Ford-Fulkerson algorithm in a simple way: always choose the shortest path from source s to the sink t, where the "shortest path" means the fewest number of edges, regardless of their capacities (i.e., each edge has the same unit weight). This algorithm runs in time $O(|V||E|^2)$.

The fastest max flow algorithm to date, an extension of the `PREFLOW-PUSH` algorithm runs in time $|V|^3$.

5.3 Solving Different Problems with Maximum Flow

5.3.1 Networks with Multiple Sources and Sinks

Flow networks with multiple sources and sinks are reducible to networks with a single source and single sink by adding a "super-sink" and "super-source" and connecting them to all sources and sinks, respectively, by edges of infinite capacities.

5.3.2 Maximum Matching In Bipartite Graphs

We will consider bipartite graphs; i.e., graphs whose vertices can be split into two subsets, L and R such that every edge $e \in E$ has one end in the set L and the other in the set R.

Matching A *matching* in a graph G is a subset M of all edges E such that each vertex of the graph belongs to at most one of the edges in the matching M.

Maximum Matching A *maximum matching* in a bipartite graph G is a matching containing the largest possible number of edges.

We turn a Maximum Matching problem into a Max Flow problem by adding a super source and a super sink, and by giving all edges a capacity of 1.

Note how the residual flow networks allow rerouting the flow in order to increase the total throughput.

5.3.3 Max Flow with Vertex Capacities

Sometimes not only the edges but also the vertices v_i of the flow graph might have capacities $C(v_i)$, which limit the total throughput of the flow coming to the vert (and, consequently, also leaving the vertex):

$$\sum_{e(u,v)\in E} f(u,v) = \sum_{e(v,w)\in E} f(u,w) \leq C(v).$$

Such a case is reduced to the case where only edges have capacities by splitting each vertex v with limited capacity $C(v)$ into two vertices v_{in} and v_{out} so that all edges coming into v go into v_{in}, all edges leaving v now leave v_{out} and by connecting the new vertices v_{in} and v_{out} with an edge $e^* = (v_{in}, v_{put})$ with capacity equal to the capacity of the original vertex v.

5.4 Applications of Max Flow Algorithm

5.4.1 Allocation: Movie Rental

Problem Assume you have a movie rental agency. At the moment you have k movies in stock, with m_i copies of the movie i. Each of n customers can rent out at most 5 movies at a time. The customers have sent you their preferences which are a list of movies they would like to see. Your goal is to dispatch the largest possible number of movies.

5.4.2 Multiple Sources and Sinks: Cargo Allocation

Problem The storage space of a ship is in the form of a rectangular grid of cells with n rows and m columns. Some of the cells are taken by support pillars and cannot be used for storage, so they have 0 capacity. You are given the capacity of every cell; cell in row r_i and column c_j has capacity $C(i, j)$. To ensure the stability of the ship, the total weight in each row r_i must not exceed $C(r_i)$ and the total weight in each column c_j must not exceed $C(c_j)$. Find how to allocate the cargo weight to each cell to maximise to total load without exceeding the limits per column, limits per row and limits per available cell.

5.4.3 Vertex Capacities: Disjoint Paths

Problem You are given a connected, directed graph G with N vertices. Out of these N vertices k are painted red, m are painted blue, and the remaining $N - k - m > 0$ of the vertices are black. Red vertices have only outgoing edges and blue vertices have only incoming edges. Your task is to determine the largest possible number of disjoint (i.e., non-intersecting) paths in this graph, each of which starts at a red vertex and finishes at a blue vertex.

6 Dynamic Programming

6.1 Foundations for Dynamic Programming

Method Build an optimal solution to the problem from optimal solutions for subproblems.

- Subproblems are chosen in a way that allows recursive construction of optimal solutions to problems from optimal solutions to smaller-size problems.
- The efficiency of DP comes from the fact that the sets of subproblems needed to solve large problems heavily overlap; each subproblem is solved only once and its solution is stored in a table for multiple uses for solving many larger problems.

6.2 Activity Selection II

Setting A list of activities $a_i, 1 \leq i \leq n$ with starting time s_i and finishing times f_i. No two activities can take place simultaneously.

Task Find a subset of compatible activities of maximal total duration.

Algorithm We start by sorting these activities by their finishing time into a non-decreasing sequence, so will assume that $f_1 \leq f_2 \leq \cdots \leq f_n$.

For $1 \leq i \leq n$, the **Subproblem** $P(i)$ is to find a subset S_i of activities $A_i = \{a_1, a_2, \ldots, a_i\}$ such that:

1. S_i consists of non-overlapping activities;

2. S_i ends with activity a_i;

3. S_i is of maximal total duration among all subsets of A_i satisfying 1 and 2.

Let $T(i)$ be the total duration of the solution S_i of the subproblem $P(i)$.
For S_1 we choose a_1l thus $T(1) = f_1 - s_1$;
Recursion: assuming that we have solved subproblems for all $j < i$ and stored them in a table, we let

$$T(i) = \max\{T(j) + f_i - s_i \mid 1 \leq j < i, f_j < s_i\}$$

Correctness Let the optimal solution of subproblem $P(i)$ be the sequence $S_i = (a_{k_1}, a_{k_2}, \ldots a_{k_{m-1}}, a_{k_m})$ where $k_m = i$.

We claim that the truncated subsequence $S' = (a_{k_1}, a_{k_2}, \ldots, a_{k_{m-1}}$ is an optimal solution to the subproblem $P(k_{m-1}$, where $k_{m-1} < i$.

If there were a sequence S^* of a larger total duration of sequence S' and also ending with activity $a_{k_{m-1}}$, we could obtain a sequence $\hat{S}$ by extending the sequence S^* with activity a_{k_m} and obtain a solution for subproblem $P(i)$ with a longer total duration than the total duration of sequence S_i, contradicting the optimally of S_i. Continuing with the solution of the problem, we now let

$$T_{\max} = \max\{T(i) \mid i \leq n\}.$$

We can now reconsturct the optimal sequence which solves our problem from the table of partial solutions, because in the i^{th} slot of the table, besides $T(i)$, we also store j such that the optimal solution of $P(i)$ extends the optimal solution of subproblem $P(j)$.

If such optimal solution ends with activity a_k, it would have been obtained as the optimal solution of problem $P(k)$.

Complexity

1. Sorting takes $O(n \lg n$

2. We need to solve n subproblems. Each subproblem requires examining the preceding subproblems and their optimal solutions. This takes $O(n^2)$.

3. We need $O(n)$ to compute $T_{\max}$ and conclude.

Thus, the overall time is $O(n^2)$.

6.3 Longest Increasing Subeqence

Setting Given a sequence of n real numbers $A[1 \ldots n]$.

Task Determine a subsequence (not necessarily contiguous) of maximum length in which the values in the subsequence are strictly increasing.

Algorithm For each $1 \leq i \leq n$ **Subproblem** $P(i)$: Find a subsequence of the sequence $A[1 \ldots i]$ of maximum length in which the values are strictly increasing and which ends with $A[i]$.

Recursion: Assume we have solved all the subproblems for $j < i$; We now look for all $A[m]$ such that $m < i$ and such that $A[m] < A[i]$;

Among those, we pick m which produced the longest increasing subsequence ending with $A[m]$ and extend it with $A[i]$ to obtain the longest increasing subsequence which ends with $A[i]$.

Correctness We claim that truncating the optimal solution for $P(i)$ will produce an optimal solution for $P(m)$ and follow a very similar proof to the activity selection problem.

Time Complexity This algorithm runs in $O(n^2)$. This problem can be done in $O(n \log n)$ time.

6.4 Integer Knapsack Problem (without Duplicates)

Setting You have n items (some of which can be identical); item I_i is of weight w_i and value v_i. You also have a knapsack of capacity C.

Task Chose a combination of available items which all fit in the knapsack and whose value is as large as possible.

Algorithm For all $1 \leq i \leq n$ and $0 \leq c \leq C$, the subproblems $P(i, c)$ is of the form

Choose from items $I_2, I_2, \ldots, I_i$ a subset which fits in a knapsack of capacity c and is of the largest possible total value.

Let $m(i, c)$ be this largest value.

- This is an example of "2D" recursion; we are filling a table of size $n \times C$, row by row.
- Fix now $i \leq n$ and $c \leq C$ and assume we have solved the subproblems for:
  1. all $j < i$ and all knapsacks of capcities from 1 to C;
  2. for i we have solved the problem for all capactities $d < c$.

We now have two options: either we take item I_i or we do not. So we look at optimal solutions $m(i-1, c-w_i)$ and $m(i-1, c)$.

$$m(i, c) = \max(m(i-1, c-w_i) + v_i, m(i-1, c))$$

Final solution will be given by $m(n, C)$.

Edge Cases

- What happens if $c - w_i < 0$? Knapsack capacity exceeded!
- What if $i - 1 < 1$? No more items to be taken!

Let $m(i, c) = -\infty$ for $c < 0$. Let $m(0, c) = 0$ for $c \geq 0$.

6.5 Balanaced Partition

Setting You have a set S of n integers.

Task Partition S into two subsets S_1, S_2 such that you minimise $|s_1 - s_2|$, where s_1 and s_2 denote the sums of the elements in each of the two subsets.

Solution Let s be the total num of all integers in the set; consider the Knapsack problem (without duplicates) with the knapsack of size $s/2$ and with each integer x_i of both size and value equal to x_i.

Claim The best packing of such knapsack produces optimally balanced partition, with S_1 being all the integers in the knapsack and S_2 all the integers left out of the knapsack.

Since $s = s_1 + s_2$ we obtain $2(\frac{s}{2} - s_1 = s - 2s_1 = s_2 - s_1$. Thus, minimising $\frac{s}{2} - s_1$ will minimise $s_2 - s_1$. So, all we have to do is find the subset of these numbers with the largest possible total sum which fits inside a knapsack of size $s/2$.

6.6 Assembly Line Scheduling

Setting Two assembly lines with workstations for n jobs.

- Executing the k^{th} job on assembly line i takes a_k^i units of time to complete $i \in \{1, 2\}, (1 \leq k \leq n)$.
- Moving the product from stations k on assembly line i to stations $k+1$ on line $(2-i)$ takes t_k^i units of time.
- Bringing an unfinished product to assembly line i takes e^i time.
- Shipping a finished product off assembly line i takes x^i time.

Task Find a *fastest way* to assemble a product using both lines as necessary.

Subproblem For $1 \leq k \leq n$ and $i \in \{1, 2\}$, the subproblem $P(i, k)$ is to find the minimal amount of time $m(i, k)$ needed to finish the first k jobs, such that k^{th} job is finished on the k^{th} workstation on the i^{th} line.

- We solve $P(1, k)$ and $P(2, k)$ by simultaneous recursion on k:
- Inital step: $m(1, 1) = e^1 + a_1^1$ and $m(2, 1) = e^2 + a_1^2$.
- Heredity step

$$m(1, k+1) = \min\{m(1, k) + a_{k+1}^1, m(2, k) + t_k^2 + a_{k+1}^1\}$$

$$m(2, k+1) = \min\{m(2, k) + a_{k+1}^2, m(1, k) + t_k^1 + a_{k+1}^2\}$$

- Finally, the overall solution is $opt = \min\{m(1, n) + x^1, m(2, n) + x^2\}$

Shortest Path Solution

- Split every station into 2 vertices, "station entry" and "station exit".
- Cost a_k^i between the entry and the exit of a station.
- Cost 0 between exit and entry of consecutive stations on the same line.

6.7 Pseudo-Polynomial Time

6.7.1 Making Change

Setting You are given n types of coin denominations of values $v_1 < v_2 < \cdots < v_n$ (all integers). Assume $v_1 = 1$, so that you can always make change for any integer amount. Assume that you have an unlimited supply of coins of each denomination.

Task Give an algorithm which makes change for any given integer amount C with as few coins as possible.

Main Idea

- Consider an optimal solution S_i for amount $i \leq C$.
- If $i > 0$, then S_i includes at least one coin, say, of denomination v_k.
- Removing this coin must produce an optimal solution for the amount $i - v_k$, $S_i - v_k$, again by our *cut-and paste argument*.
- We do not know which coins S_i includes, so we try all the available coins and then pick k for which $Si - v_k$ uses the fewest number of coins.

Algorithm

- For $O \leq i \leq C$, subproblem $P(i)$ is to make change for amount i with as few coins as possible. Let $m(i)$ be the number of coins required.
- If $i = 0$ the solution is trivial: you don't need any coin, $m(0) = 0$.
- Assume optimal solution for amounts $j < i$ and find an optimal solution for amount i. That is, $m(i) = \min\{m(i - v_k) + 1 \mid 1 \leq k \leq n, i - v_k \geq 0\}$.
- Don't forget the condition $i - v_k \geq 0$ or else define $m(i) = \infty$ for $i < 0$.

Complexity The time complexity of our algorithm is $\Theta(nC)$.
Length of input: $O(\lg C + \lg v_1 + \lg v_2 + \cdots \lg v_n) = O(n \lg C)$.
Our algorithm is NOT polynomial in the length of the input! But this is the best that we have at out disposal...
Because *Making Change* is an NP-Complete Problem.

6.7.2 Integer Knapsack Problem with Duplicates

Setting You have n types of items; all items of kind i are identical and of weight w_i and value v_i. You also have a knapsack of capacity C.

Task Choose a combination of items which all fit in the knapsack and whose value is as large as possible. You can take any number of items of each kind.

Solution DP recursion on the capacity C of the knapsack. We build a table of optimal solutions for all knapsacks of capacities $i \leq C$. Assume we have solved the problems for all knapsacks of capacities $j < i$. We now look at optimal solutions $m(i - w_m$ for all knapsacks of capacities $i - w_m$ for all $1 \leq m \leq n$. Chose the one for which $m(i - w_m) + v_m$ is the largest. Add to it the item m to obtain a packing of a knapsack of size i of the highest possible value. Thus, $m(i) = \max\{m(i - w_m) + v_m : 1 \leq m \leq n\}$. After C many steps we obtain $m(C)$ which is what we need.

Again, our algorithm is NOT polynomial in length of the input.

6.7.3 Pseudo-Polynomial Time

Consider a problem with numerical input of magnitude N and optionally non-numerical input size n. A pseudo-polynomial algorithm to solve this problem is an algorithm that runs in time $O(P(n)P'(N))$ where P and P' are polynomials.

Example: Making Change Numerical input: the denominations $v_1, \ldots, v_n$ and the target C. So the magnitude is $N = |C| + \sum |v_i|$. All input is numerical and our proposed algorithm runs in $O(N)$ so it is pseudo-polynomial.

6.8 Shortest Path Algorithms

6.8.1 Bellman-Ford: Shortest Paths with Negative Weights

Setting A directed weighted graph $G = (V, E)$ with weights which can be negative, but without cycles of negative total weight and a vertex $s \in V$.

Task Find the shortest path from vertex s to every other vertex t.

Solution Since there are no negative weight cycles, the shortest path cannot contain cycles, because a cycle can be excised to produce a shorter path. Thus, every shortest path can have at most $|V| - 1$ edges.

Subproblems: For every $v \in V$ and every $i, (1 \leq i \leq n-1)$, let $opt(i, v)$ be the length of a shortest path from s to v which contains at most i edges. Our goal is to find for every vertex $t \in G$ the value of $opt(n-1, t)$ and the path which achieves such a length.

Note that if the shortest path from a vertex v to t is $(v, p_1, p_2, \ldots, p_k, t)$ then $(p_1, p_2, \ldots, p_k, t)$ must be the shortest path from p_1 to t and $(v, p_1, p_2, \ldots, p_k)$ must also be the shortest path from v to p_k.

Let us denote the length of the shortest path from s to v among all paths which contain at most i edges by $opt(i, v)$ and let $pred(i, v)$ be the immediate predecessor of vertex v on such shortest path.
Recursion:

$$opt(i, v) = \min(opt(i-1, v), \min_{p \in V}\{opt(i-1, p) + w(e(p, v))\};$$

$$pred(i, v) = \begin{cases} pred(i-1, v) & \text{if } \min_{p \in V}\{opt(i-1, p) + w(e(p, v))\} \geq pred(i-1, v) \\ \arg\min_{p \in V}\{opt(i-1, p) + w(e(p, v))\} & \text{otherwise} \end{cases}$$

(here $w(e(p, v))$ is the weight of the edge $e(p, v)$ from vertex p to vertex v.) Algorithm produces shortest paths from s to every other vertex in the graph.

Time Complexity Computation $opt(i, v)$ runs in time $O(|V| \times |E|)$, because $i \leq |V| - 1$ and for each v, min is taken over all edges $e(p, v)$ incident to v; thus in each round all edges are inspected.

6.8.2 Floyd-Warshall

Let again $G = (V, E)$ be a directed weighted graph where $V = \{v_1, v_2, \dots, v_n\}$ and where weights $w(e(v_p, v_q))$ of edges $e(v_p, v_q)$ can be negative, but there are no negative weight cycles. We can use a somewhat similar idea to obtain the shortest paths from every vertex v_p to every vertex v_q (including back to v_p).

Let $opt(k, v_p, v_q)$ be the length of the shortest path from a vertex v_p to a vertex v_q such that all intermediate verticies are among verticies $\{v_1, v_2, \dots, v_k\}, (1 \leq k \leq n)$. Then

$$opt(k, v_p, v_q) = \min\{opt(k-1, v_p, v_q), opt(k-1, v_p, v_k) + opt(k_1, v_k, v_q)\}$$

Thus, we gradually relax the constraint that the intermediate vertices have to belong to $\{v_1, v_2, \dots, v_k\}$. The algorithm runs in time $|V|^3$.

7 Reductions

7.1 Decision Problems

A *decision problem* is a problem with a YES/NO answer.

Certificates and Counter-Example For a given problem P and a given instance x,

- a certificate for x is data that lets us verify easily that $P(x)$ =YES.
- a counter-example for x is data that lets us verify easily that $P(x)$ =NO.

Polynomial Time Algorithms A decision problem A is in polynomial time if there exists a polynomial time algorithm that solves it. An algorithm runs in polynomial time for every input if it terminates in polynomially many steps in the length of the input (i.e. $T(n) = O(n^k)$ where k is a natural number and n is the size of the input). We denote this by $A \in \mathbf{P}$.

Input The length of an input is the number of symbols needed to describe the input precisely.

Reductions Decision problem U is reducible to dec. prob. V if there is a function f such that

1. f maps instances of U into instances of V;
2. For every instance x of U, $U(x)$ is true iff $V(f(x))$ is true.

If f is commutable in polynomial time then U is polynomially reducible to V.

Polynomial Reduction from SAT to 3SAT Every instance of SAT (Boolean SATisfiability Problem) is polynomially reducible to an instance of 3SAT. (See Video/Slide)

7.2 Linear Programming

Variables

$$x_j \text{ for } 1 \leq j \leq n$$

Objective

$$\text{maximise or minimise } \sum_{j=1}^{n} c_j x_j$$

Constraints

$$\sum_{j=1}^{n} (a_{ij} x_j) \mathcal{R}_i b_i, \qquad \text{for } 1 \leq i \leq m \text{ with } \mathcal{R}_i \in \{\leq, =, \geq\}$$

A feasible solution is a variable assignment satisfying all constraints. An optimal solution is a feasible solution satisfying the objective.

Canonical Form

- Objective: maximise $\sum_{j=1}^{n} c_j x_j$
- Constraints: $\sum_{j=1}^{n} a_{ij} x_j \leq b_i$, for $1 \leq i \leq m$ and $x_j \geq 0$ for $1 \leq j \leq n$.

Matrix Form To specify a linear programming problem we can simply provide a triplet $(A, \mathbf{b}, \mathbf{c})$ where A is a matrix and $\mathbf{b}, \mathbf{c}$ are column vectors (see slide).

Standard Form

- maximise $z = \mathbf{c}^T \mathbf{x}$
- subject to the constraints $A\mathbf{x} + I\mathbf{s} = \mathbf{b}$ and $\mathbf{x} \geq 0$ and $\mathbf{s} \geq 0$.

$\mathbf{s}$ are the slack and surplus variables that are used to transform constraints using inequalities into equality constraints.

Transformations Any LP can be transformed into a canonical form or into standard form if needed. In general a Linear Program does not necessarily produce the non-negativity constraints for all variables. However, in the standard form such constraints are required for all of the variables. This is not a problem because each occurrence of an unconstrained variable x_j can be replaced by the expression $x'_j - x_j*$ where x'_j, x_j* are new variables satisfying the constraints $x'_j, x_j* \geq 0$. Similarly constraints of the form $|A\mathbf{x}| \leq \mathbf{b}$ can be replaced to two linear constraints: $A\mathbf{x} \leq \mathbf{b}, -A\mathbf{x} \leq \mathbf{b}$.

Algorithms

- Simplex (1947): Exponential runtime in the worst case, very efficient in practice
- Ellipsoid Method (1979): Polynomial Algorithm $O(n^6 L)(n$ variables, input of size $L)$
- Interior Point algorithms: Worst-case $O(n^{3.5} L^2 \lg L \lg \lg L)$, farily efficient in practice

Variants

- Integer Linear Programs (ILP) (NP-complete)
- Mixed Integer Linear Programs (continuous and integer variables)
- 0-1 Linear Programming (variables are $\in \{0, 1\}$)

7.2.1 Decide Diet - Linear Programming

Setting You are given a list of food sources $f_1, f_2, \ldots f_n$ for each source f_i you are given:

- its price per gram p_i
- the number of calories c_i per gram and
- for each of 13 vitamins $V_1, V_2, \ldots, V_{13}$ you are given the content $v(i, j)$ of milligrams of vitamin V_j in one gram of food source f_i.

For each vitamin V_j, you are given the recommended daily intake of w_j milligrams.

Task Find a combination of quantities of food sources such that:

- the total number of calories in all of the chosen food is equal to a recommended daily value of 2000 calories
- the total intake of each vitamin V_j is at least the daily intake of w_j milligrams for all $1 \leq j \leq 13$
- the price of all food per day is as low as possible.

Solution To obtain the corresponding constraints let us assume that we take x_i grams of each food source f_i for $1 \leq i \leq n$. Then:

- the total number of calories must satisfy $\sum_{i=1}^{n} x_i c_i = 2000$;
- for each vitamin V_j the total amount in all food must satisfty

$$\sum_{i=1}^{n} x_i v(i,j) \geq w_j \quad (1 \leq j \leq 13);$$

- an implicy assumption is that all quantities must be non-negative $x_i \geq 0, 1 \leq i \leq n$.

Our goal is to minimise the objective function which is the total cost $y = \sum_{i=1}^{n} x_i p_i$. Note that here all the equalities and inequalities, as well as the objective function are linear.

7.2.2 Infrastructure Politics - Integer Programming

Setting You are the (Shadow?) Treasurer and you want to make certain promises to the electorate that will ensure that your party will win in the forthcoming elections. You promise that you will build

- a certain number of bridges, each 3 billion a piece. Each bridge you promise brings you 5% of city votes, 7% of suburban votes and 9% of rural votes.
- a certain number of rural airports, each 2 billion a piece. Each rural airport you promise brings you no city votes, 2% of suburban votes and 15% of rural votes.
- a certain number of olympic swimming pools each a billion a piece. Each olympic swimming pool promised brings you 12% of city votes, 3% of suburban votes and no rural votes.

Task In order to win, you have to get at least 51% of each of the city, suburban and rural votes. Win the election by cleverly making a promise that appears to blow as small hole in the budget as possible.

Solution Let the number of bridges, airports and swimming pools to be x_b, x_a, x_p respectively. The problem amounts to minimising the objective $y = 3x_b + 2x_a + x_p$, while making sure that the following constraints are satisfied.

$$\begin{aligned} 0.05x_b + 0.12x_p &\geq 0.51 && \text{(securing majority of city votes)} \\ 0.07x_b + 0.02x_a + 0.03x_p &\geq 0.51 && \text{(securing majority of suburban votes)} \\ 0.09x_b + 0.15x_a &\geq 0.51 && \text{(securing majority of rural votes)} \\ x_b, x_a, x_p &\geq 0. \end{aligned}$$

This is an example of Integer Linear Programming which is much harder than the "plain" Linear Programming and is in fact NP hard!

7.3 NP Completeness

A decision problem $A(x)$ is in non-deterministic polynomial time, denotes by $A \in \mathbf{NP}$, if:

1. there exists a problem $B(x, y)$ such that for every input $x, A(x)$ is true just in case there exists y such that $B(x, y)$ is true; and

2. such that the truth of $B(x, y)$ can be verified by an algorithm running in polynomial time in the length of x only.

We call y a certificate of x.

NP-hardness A problem is NP-hard if any problem in NP is reducible to it. I.e., a problem P is NP-hard if for any other problem P' that is in the class NP, there exists a polynomial reduction $f_{P'}$ from P' to P. A problem is NP-complete if it is both NP-hard and in the class NP.

Proving NP completeness Sometimes the distinction between a problem in P and an NP complete problem can be subtle!

in P	**NP complete**
Given a graph G and two vertices s and t, is there a path from s to t of length **at most** K?	Given a graph G and two vertices s and t, is there a simple path from s to t of length **at least** K?
Given a propositional formula in CNF form such that every clause has at most **two** propositional variables, does the formula have a satisfying assignment?	Given a propositional formula in CNF form such that every clause has at most **three** propositional variables, does the formula have a satisfying assignment?
Given a graph G, does G have a tour where every **edge** is traversed exactly once? (An Euler tour.)	Given a graph G, does G have a tour where every **vertex** is visited exactly once? (A Hamiltonian cycle.)

Theorem Let U be an NP-hard problem and let V be another decision problem. If U is polynomially reducible to V then V is also NP-hard.

NP Hard Optimisation If an optimisation problem is NP-hard, we do not try to solve it exactly, but instead, try to find a feasible (i.e., P time) algorithm which produces a solution that is not too bad. Examples (extra info in slides):

- Vertex Cover: We use an approximation algorithm that always produces a covering set with at most twice the number of the smallest vertex cover.

- Metric Traveling Salesman: Has an approximation algorithm producing a tour of total length at most twice the length of the optimal, minimal length tour.

Cook's Theorem Every NP problem is polynomially reducible to the SAT problem.

NP Complete Examples 3SAT, Travelling Salesman, Register Allocation, Set Cover,...