3121 Notes

Chris Wong

1 Divide and Conquer

1.1 Binary Search

- **Divide:** Test the midpoint of the search range.
- **Conquer:** Search one side of the midpoint recursively.
- **Combine:** Pass the answer up to the recursion tree.

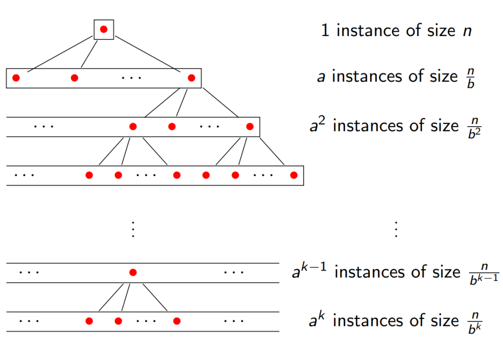
Binary Search can be applied to solve optimisation problems by reducing it to a decision problem, if the structure of the decision problem can be applied *monotonically* to the optimisation problem. If every point $P(k)$ is yes then $P(k+1)$ will also be yes, and if $P(k')$ is no, then $P(k'-1)$ will also be no.

1.2 Recurrences

A general recurrence relation for divide and conquer algorithms is formulated generally as

- Suppose a divide-and-conquer algorithm:
  - Reduces a problem of size n to a subproblems of *smaller size* $\frac{n}{b}$
  - with overhead cost of $f(n)$ to *split* up the problem and *combine* the solutions from these smaller problems.
- A recurrence relation follows from this

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



1.3 Master Theorem

We define the critical exponent $c^* = \log_b a$ and the critical polynomial n^{c^*}.

- If $f(n) = O(n^{c^*-\epsilon})$, then $T(n) = \Theta(n^{c^*})$.
- If $f(n) = \Theta(n^{c^*})$, then $T(n) = \Theta(n^{c^*} \log n)$.
- If $f(n) = \Omega(n^{c^*+\epsilon})$, and the following holds for some $\epsilon > 0$, and for some $k < 1$ and some n_0
$$af(\frac{n}{b}) \leq kf(n)$$
holds for all $n > n_0$ then $T(n) = \Theta(f(n))$.

1.4 Multiplying Integers

To multiply two input numbers A and B we apply the divide-and-conquer framework as follows.

- Split the two numbers A and B into halves:
  - A_0, B_0 - the least significant $\frac{n}{2}$ bits
  - A_1, B_1 - the most significant $\frac{n}{2}$ bits

$$A = A_1 2^{\frac{n}{2}} + A_0$$
$$B = B_1 2^{\frac{n}{2}} + B_0$$

- AB can be calculated from the following

$$AB = A_1B_1 2^n + (A_1B_0 + B_1A_0)2^{\frac{n}{2}} + A_0B_0$$

- Compute
  - $X = A_0B_0$
  - $Y = A_0B_1$
  - $Z = A_1B_0$
  - $W = A_1B_1$
- Now compute $W2^n + (Y+Z)2^{\frac{n}{2}} + X$
  - W shifted left by n bits
  - Y and Z shifted left by $\frac{n}{2}$ bits
  - X with no shift

We then construct the following recurrence relation

$$T(n) = 4T(\frac{n}{2}) + cn$$

applying the master theorem where $n^{\log_2 4} = n^2$ such that $n = O(n^{2-\epsilon})$ and so Case 1 applies such that $T(n) = \Theta(n^2)$.

1.5 Karatsuba Trick

We can apply a famous modification rearranging our previous AB equation to be as follows:

$$AB = A_1B_1 2^n + ((A_1+A_0)(B_1+B_0) - A_1B_1 - A_0B_0)2^{\frac{n}{2}} + A_0B_0$$

- Now we compute
  - $X = A_0B_0$
  - $W = A_1B_1$
  - $V = (A_1 + A_0)(B_1 + B_0)$
- Now our recurrence reduces to

  $$T(n) = 3T(\frac{n}{2}) + cn$$

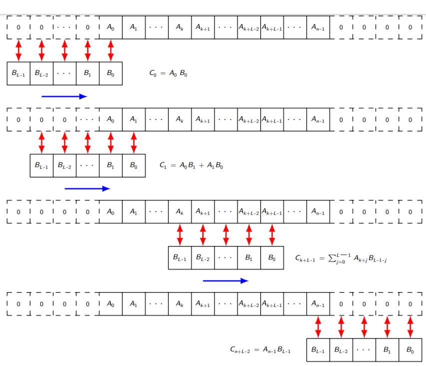
  such that our $n^{\log_2 3} = n^{1.58}$ which is Case 1 of MT, such that $T(n) = \Theta(n^{1.58})$.

1.6 Convolutions

We observe that multiplying two polynomials AB involves multiplying every pair A_iB_j, but more importantly the significance of the coefficient sequence appears from the fact that

- The coefficient $c_0x^{k'}$ is the sum of the total amount of x^k between A and B that can add up to k'

  $$C_t = \sum_{i+t=t} A_iB_j$$



1.7 FFT

Allows us to multiply two polynomials in $O(n \log n)$ time and consequently allows computing of convolutions in that time aswell.

- $T(m) = 2T(\frac{m}{2}) + cm$
- Case 2 of MT gives $T(m) = \Theta(m \log m)$
- **DFT:**

  $$X[k] = \sum_{j=0}^{N-1} x[j] \cdot e^{-i\frac{2\pi}{N}kj}$$

2 Greedy

A greedy algorithm chooses the locally optimal choice at each stage using a heuristic that will lead to a globally optimal outcome.

2.1 Greedy Stays Ahead

Prove that at every stage, no other sequence of choices could do better than our proposed algorithm

- Consider a metric g_i chosen by our greedy solution G and o_i chose by the optimal solution O.
- Show that $g_1 \geq o_1$ (inequality may vary on heuristic)
- Assume this holds up to k decisions.
- Show that $g_{k+1} \geq o_{k+1}$
- Therefore $g_m \geq o_m$ and our greedy solution will always be as optimal as O

2.2 Exchange Argument

Consider an alternative solution, and gradually transform it to the solution found by our proposed algorithm without making it any worse,

- Consider an alternative solution $O = \{o_1, \ldots, o_m\}$ and our greedy solution $G = \{g_1, \ldots, g_m\}$.
- Assume G and O differs in at least one place. Assume $k \leq m$ is the first index in which $g_k \neq o_k$.
- Form a new solution $O' = \{o_1, \ldots, o_{k-1}, g_k, o_{k+1}, \ldots, o_m\}$
- Show O' is at least as optimal as O. Therefore we can continue to exchange until O' transforms to G.

2.3 Tarjan's Algorithm

Algorithm to find all strongly connected components of a graph.

- Run a DFS and maintain an explicit stack (seperate from DFS stack)
- Mark vertices pushed onto the stack as "in-stack"
- If we reach a vertex v that is already "in-stack" then each item before v
  - can be reached from v (following path of DFS)
  - can reach v (using the same edge encountered)

- Pop everything before and including v, and add to v's SCC

2.4 Condensation Graph

In a graph $G = (V, E)$ if there exists strongly connected components we can transform said graph into a condensation graph $\Sigma_G = (C_g, E^*)$ such that all SCG are transformed into vertices. Note: Σ_G is acyclic

$$E^* = \{(C_{u_1}, C_{u_2}) | (u_1, u_2) \in E, C_{u_1} \neq C_{u_2}\}$$

- Vertices are the strongly connected components of G

- Edges correspond to edges of G not within a SCG, duplicates ignored

2.5 Topological Sorting

In a case where a directed graph G has no cycles, we can order vertices of G as follows:

- Let $G = (V, E)$ be a directed graph. A topological sort of G is a linear ordering of it's vertices $\sigma : V \rightarrow \{1, \ldots, n\}$ such that if there exists an edge $(v, w) \in E$ then v precedes w in the ordering. e.g $\sigma(v) < \sigma(w)$

- Maintain
  - A list L of vertices
  - An array D consisting of the in-degrees of vertices
  - A set S of vertices with no incoming edges

- While set S is not empty, select a vertex u with $D[u] = 0$ in the set
  - Remove it from S and append it to L
  - Then, for every outgoing edge $e = (u, v)$ from the vertex, remove the edge from the graph, and decrement $D[v]$ accordingly. If $D[v]$ is now 0, insert v into S

- Algorithm runs in $O(V + E)$

2.6 Dijkstra's Algorithm

Dijkstra's Algorithm: Correctness 128

First, we will prove the correctness of Dijkstra's algorithm.

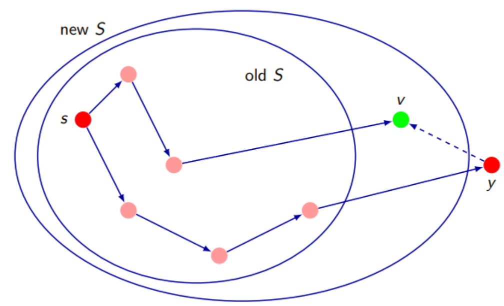
Claim

Suppose v is the next vertex to be added to S. Then d_v is the length of the shortest path from s to v.

Proof

- d_v is the length of the shortest path from s to v using only intermediate vertices in S. Let's call this path p.
- If this were not to be the shortest path from s to v, there must be some shorter path p' which first leaves S at some vertex y before later reaching v.

Dijkstra's Algorithm: Correctness 129

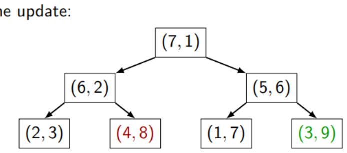


Augmented Heaps 142

- We will use a heap represented by an array $A[1..n]$; the left child of $A[j]$ is stored in $A[2j]$ and the right child in $A[2j+1]$.
- Every element of A is of the form $A[j] = (i, d_i)$ for some vertex i. The min-heap property is maintained with respect to the d-values only.
- We will also maintain another array $P[1..n]$ which stores the position of elements in the heap.
- Whenever $A[j]$ refers to vertex i, we record $P[i] = j$, so that we can look up vertex i using the property $A[P[i]] = (i, d_i)$.

Augmented Heaps 145

Example. Update d[3] from 9 to 5.
Before the update:



j	1	2	3	4	5	6	7	
$A[j]$	7	6	5	2	4	1	3	i
	1	2	6	3	8	7	9	d_i

i	1	2	3	4	5	6	7
$P[i]$	6	4	7	5	3	2	1

2.7 Minimum Spanning Trees

$G = (V, E)$ is a connected graph and each edge e in E has a non-negative length

- A spanning tree T of G is any tree which is a subgraph of G on the same vertex set V
- The weight of a tree T is the sum of all edge lengths in T
- A minimum spanning tree T of a connected graph G is a tree such that weight is minimised

2.8 Kruskal's Algorithm

An algorithm to find the minimum spanning tree as follows:

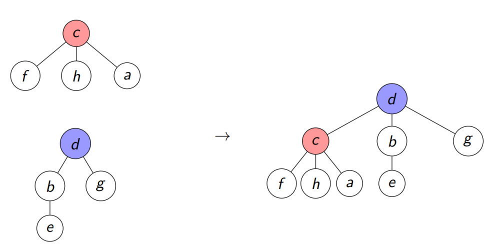
- Sort the edges E in increasing order by weight
- Sweep the edges and add it to the graph if
  - An edge e is added if its inclusion does not introduce a cycle in the graph constructed thus far, or discarded otherwise.
- Algorithm terminates when $n-1$ edges have been added

2.9 Union Find

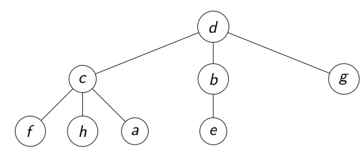
To efficiently determine if adding an edge will create a cycle we use a union find data structure which handles the following operations.

- Maintain a forest, where each tree represents one of our disjoint sets
- Union(A, B): Merges the sets A and B into a single set $A \cup B$

  We place the edge between the roots of the trees containing a and b.



- Find(A): Finds the representative of the set containing A
  - Find(A) = Find(B) if and only if A and B are in the same set
  - Find(A) can change as a result of union operations
- Find(A) will involve starting at a and traversing the edge to the parent, until there is no parent.
- Union(A,B) will run Find(A) and Find(B) to determine if they're disjoint, and place an edge between the roots of A and B the parents will be arbitrary for now.
- This will take $O(h)$, to ensure our trees are balanced so that $O(h) = \log h$ we will use the size heuristic.
  - For each vertex, maintain not only the parent but the size of the subtree rooted at that vertex



v	a	b	c	d	e	f	g	h
parent(v)	c	d	d	–	b	c	d	c
subtree(v)	1	2	4	8	1	1	1	1

- Now our Union(A,B) will decide the parent when adding an edge between the roots of A and B as follows:
  - If subtree(c) $\geq$ subtree(d), then parent(d) = c
  - If subtree(d) $\geq$ subtree(c), then parent(c) = d
- This will maintain the property that if Union(A,B) makes B the new parent of A then subtree(B) ≥ 2 subtree(A)
- Therefore using this for Kruskal's will be $O(m \log n)$ which is the same as sorting it.

2.10 k-clustering

Given an instance of a complete graph G with weighted edges, we want to partition vertices of G into disjoint subsets such that two points belonging to different sets are as large as possible.

- Sort the edges in increasing order and perform Kruskal's algorithm
- Stop when there are k connected components
- Since we have a complete graph the algorithm will run in $O(n^2 \log n)$

3 Flow Networks

A flow network $G = (V, E)$ is a directed graph in which each edge $e = (u, v) \in E$ has positive integer capacity $c(u, v) > 0$. There are two distinguished vertices: a source s and a sink t. No edge enters the source, and no edge leaves the sink.

A flow in G is a function $f : E \to [0, \infty), f(u, v) \geq 0$ that satisfies:

- Capacity constraint: for all edges $e = (u, v) \in E$ we need

  $$f(u, v) \leq c(u, v),$$

  i.e the flow through any edge does not exceed its capacity

- Flow conservation: for all vertices $v \in V \setminus \{s, t\}$ we require

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

  i.e the flow into any vertex (excluding source and sink) equals the flow out of that vertex

3.1 Residual Flow Network

Given a flow in a flow network, the residual flow network is the network made up of the leftover capacities.

- Suppose the original flow network has an edge from v to w with capacity c, and that f units of flow are being send through this edge.
- The residual flow network has two edges
  - An edge from v to w with capacity $c - f$
  - An edge from w to v with capacity f
- This represents the amount of additional flow that can be sent in each direction, note: sending flow on the edge from w to v counteracts the assigned flow from v to w

An augmenting path is a path from s to t in the residual flow network

3.2 Ford-Fulkerson Algorithm

An algorithm that will find the max flow in a flow network

- Keep adding flow through new augmenting paths as long as its possible
- When there are no more augmenting paths, the largest possible flow is achieved in the network
- Runs in $O(E|F|)$ time

3.3 Flow Network Cuts

A cut is defined as any partition of the vertices of the underlying graph into two subsets S and T such that

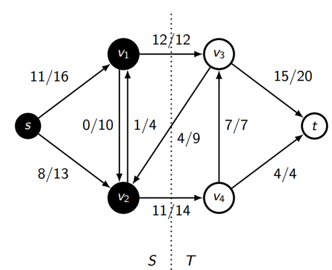
- $S \cup T = V$
- $S \cap T = \varnothing$
- $s \in S$ and $t \in T$

The capacity of $c(S, T)$ of a cut (S, T) is the sum of capacities of all edges leaving S and entering T

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

The flow $f(S, T)$ through a cut (S, T) is the total flow through edges from S to T minus the total flow through edges from T to S

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



Question. What is the capacity of this cut? 12 + 14 = 26.
Question. What is the flow across this cut? (12+11) - (4) = 19.

3.4 Max-flow Min-cut

The maximum amount of flow in a flow network is equal to the capacity of the cut of minimum capacity.

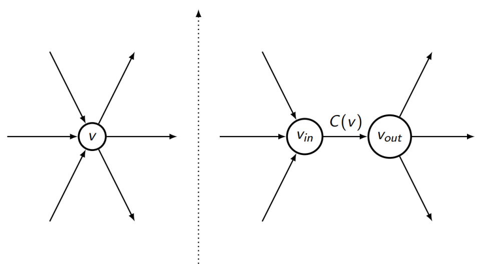
3.5 Edmonds-Karp

Edmonds-Karp is a modification of Ford-Fulkerson: EK always chooses the augmenting path of fewest edges.

- EK runs in $\min(O(VE^2), O(E|F|))$

3.6 Applications of Flow Networks

- 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-source" and "super-sink"
- Networks with vertex capacities
  - Suppose vertex v has capacity $C(v)$
  - Split v into two vertices v_{in} and v_{out}
  - Attach all of v's incoming edges to v_{in} and all of its outgoing edges from v_{out}
  - Connect v_{in} and v_{out} with an edge of capacity $C(v)$



3.7 Bipartite Matchings

A matching in a graph $G = (V, E)$ is a subset $M \subseteq E$ such that each vertex of the graph belongs to at most one edge in M

A maximum matching in G is a matching containing the largest possible number of edges.

- Create two new vertices s and t
- Construct an edge from s to each vertex in A, and from each vertex in B to t
- Add existing edges from A to B
- Assign capacity 1 to all edges
- Run EK algorithm

4 Dynamic Programming

A "smart" brute-force algorithm that reduces redundant calculations by building off of already computed sub-problems. DP is characterised by overlapping subproblems unlike DNC's disjoint subproblems.

- A dynamic programming algorthm consists of three parts:
  - A definition of the subproblem including any parameters
  - A recurrence relation, determines how the solutions to smaller subproblems are combined to solve a larger subproblem
  - Base cases, trivial subproblems those the recurrence is not requried
- Order of Computation
  - To ensure our dependencies for each subproblem are solved before solving the larger subproblem we need an order of computation that ensures our dependencies will all be defined beforehand
- Time Complexity
  - This is given by multiplying the number of subproblems by the time taken to solve a subproblem
- Overall Answer
  - The overall problem that will give us the answer, usually given by $\text{opt}(i, j)$

4.1 Directed Acyclic Graphs

We can model a DP problem as a DAG where our vertices are the subproblems and the edges represent dependencies, in particular we use a topological ordering to ensure that this will work.

For example finding the shortest path in a DAG

- **Subproblem:** Let path(t) be the shortest path from s to t
- **Recurrence:** path(t) =

$$\min\left\{\text{path}(v) + w(v,t) \quad \text{if } (v,t) \in E\right.$$

- **Base Case:** path(s) = 0
- **Order of computation:** Topological order
- **Overall answer:** list of path(t)
- **Time Complexity**: O(n + m) subproblems and O(1) to solve each subproblem

4.2 Bellman-Ford

Given a directed weighted graph $G = (V, E)$ with edge weights $w(e)$ which can be negative, but without cycles of negative total weight. We can find the weight of shortest path from vertex s to every other vertex t. (Dijkstra's will not work here as we include negative edge weights)

- **Subproblem:** Let path(i, t) be the length of a shortest path from s to t containing at most i edges
- **Recurrence:** opt(i, t) =

$$\min\left\{\text{path}(i-1, v) + w(v,t) \quad \text{if } (v,t) \in E\right.$$

- **Base Cases:** path(i, s) = 0, and for $t \neq s$, path(0, t) = ∞
- **Order of computation:** Increasing order of i, any order of t
- **Overall answer:** List of values path(n - 1, t) over all vertices t
- **Time Complexity:** $O(nm) + O(1) = O(nm)$ (nm subproblems and $O(1)$ to compute each)

4.3 Floyd-Warshall

Given a directed weighted graph $G = (V, E)$ with edge weights $w(e)$ which can be negative but without cycles of negative total weight. We can find the shortest path from every vertex s to every other vertex t

- **Subproblem:** path(i, j, k) the weight of a shortest path from v_i to v_j using only $v_1, \dots, v_k$ as intermediate vertices
- **Recurrence:** path(i, j, k) =

$$\min\begin{cases}\text{path}(i,j,k-1)\\ \text{path}(i,k,k-1) + \text{path}(k,j,k-1)\end{cases}$$

- **Base Cases:** path(i, j, 0) =

$$\begin{cases}0 & \text{if } i = j\\ w(i,j) & \text{if } (v_i, v_j) \in E\\ \infty & \text{Otherwise}\end{cases}$$

- **Order of computation:** Increasing order of k (any order of i and j)
- **Overall answer:** The table of values path(i, j, n)
- **Time Complexity:** $O(n^3)$ subproblems each taking $O(1)$ for a total of $O(n^3)$

Rabin-Karp

The Rabin-Karp algorithm utilises hashing to speed up average case performance.

$$H(A_{s+1}) = d \cdot H(A_s) - d^m a_s + a_{s+m} \bmod p$$

The hashing multiplying descending powers of the alphabet size to each alphabetical index of the characters.

4.4 KMP Algorithm

We can find the amount of string matches using the KMP algorithm

- Maintain pointers l and r into the text, which record the left and right boundaries of the current partial match
  - Initially, $l = 1$ and $r = 0$
  - Use $w = r - l + 1$ as shorthand for length of the current partial match
- Compare the next character of the text a_{r+1} to b_{w+1}, if they agree extend the partial match
- Otherwise, shorten the partial match reduce w to $\pi(w)$ by increasing l by the appropriate amount.
- If the characters agrees increase r by one and move on.
- If a match of length 0 can't be extended, increase both l and r by one.
- If the match length w reaches m report a match, then reduce w to $\pi(m)$ by increasing l.

Time Complexity will be $O(n)$ since our left and right pointers only ever move forward and at worst will each move n times forward.

4.5 Failure Function

A function that determines the longest prefix-suffix at any particular length of the string.

- **Subproblems:** Let $\pi(k)$ be the lenght of the longest prefix-suffix of B_k
- **Recurrence:** $\pi(k+1)$=

$$\min\begin{cases}\pi(k)+1 & \text{if } b_{k+1} = b_{\pi(k)+1}\\ \pi(\pi(k))+1 & \text{else if } b_{k+1} = b_{\pi(\pi(k))+1}\\ \pi(\pi(\pi(k)))+1 & \text{else if } b_{k+1} = b_{\pi(\pi(\pi(k)))+1}\\ \dots & \dots\end{cases}$$

- **Base Case:** $\pi(1) = 0$
- **Order of computation:** Increasing order of k
- **Overall answer:** the entire list of values of $\pi(k)$

- **Time Complexity:** $O(m)$ by similar two-pointer argument

We can also interpret this as a finite automata

k	Matched	x	y	z
0	$\emptyset$	1	0	0
1	x	1	2	0
2	xy	3	0	0 .
3	xyx	1	4	0
4	$xyxy$	5	0	0
5	$xyxyx$	1	4	6
6	$xyxyxz$	7	0	0
7	$xyxyxzx$	1	2	0

5 Complexity Theory

5.1 Polynomial Time Algorithms

A algorithm is polynomial time if for every input it terminates in polynomially many steps in the length of the input.

- The length of input refers to the number of symbols required to encode our input.
  - If an input x is an integer, x is encoded in bits on a machine such that the length of an input integer x is $\log_2(x)$ since this the amount of bits required to encode x as an integer. e.g $|16| = \log_2 16 = 4$ bits meaning $2^4 = 16$ s.t $x = 2^{\log_2 x}$
- For a weighted graph $G = (V, E)$ with weights $\leq W$ it can run in
  - $O(E \log W)$ for an adjacency list
  - $O(V^2 \log W)$ for an adjacency matrix

5.2 P vs NP

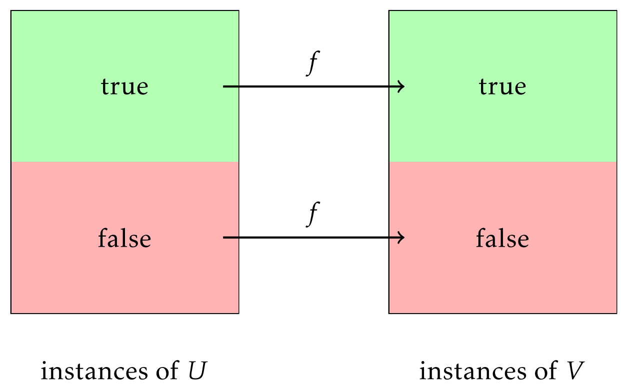
We define P and NP as complexity classes that categorise algorithms in terms of their difficulty. (oversimplification)

- A decision problem $A(x)$ is in class P if there exists a polynomial time algorithm which solves it.
- A decision problem $A(x)$ is in class NP if there is a problem $B(x, y)$ such that
  - For every input x, $A(x)$ is true if and only if there is some y for which $B(x, y)$ is true
  - The truth of $B(x, y)$ can be verified by a polynomial time algorithm in the length of x
  - In other words, it can verify a proposed solution to the problem in polynomial time.
- y is denoted as the certificate and B is denoted as the certifier.

5.3 Reductions

Reductions map a problem to another problem, however note a mapping between two problems need not be surjective i.e we may only map to specific instances of the old problem.

- Let U and V be two decision problems. U is polynomially reducible to V if and only if there exists a function $f(x)$ such that:
  - f(x) maps instances of U into instances of V
  - f maps YES instances of U to YES instances of V and NO instances of U to NO instances of V
  - f(x) is computable by a polynomial time algorithm

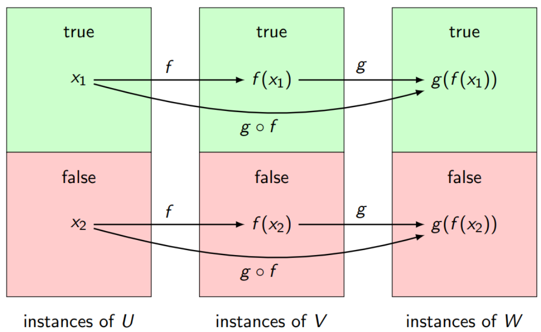


- To prove a valid reduction we must prove that:
  - If x is a YES instance of A, then $f(x)$ is a YES instance of B
  - If $f(x)$ is a YES instance of B, then x is a YES instance of A

5.4 NP-Hardness / Completeness

- A decision problem V is NP-hard if every other NP problem is polynomially reducible to V. **Cook's Theorem** states that SAT is NP-hard
- A decision problem is NP-complete, if it is both in class NP and NP-H
  - SAT is in NP and from Cook's theorem NP-H hence its NP-C

Reductions are transitive such that if there is a NP-C problem V and a NP problem U if there is a polynomial reduction of V to U it means V is at least as hard as U, and correspondingly V is at least as every problem in NP s.t U is at least as hard as every problem in NP. i.e a reduction from an NP problem to V can also be mapped to U since we have a function f : $V \rightarrow U$ from our reduction.



5.5 Linear Programming

A way to approach optimisation problems where we can create the "recipe" for a linear programming as follows:

- A set of **real-number** variables
- A set of linear inequalities known as constraints
- An objective function that we maximise / minimise
- Linear program will now compute the objective function in polynomial time
- Equality constraints are mapped by a pair of inequality constraints e.g $x_L + x_T \leq 3$ and $-x_L - x_T \leq -3 \rightarrow x_L + x_T = 3$
- Minimisation problems can be converted to maximisation by negating the objective function
- $\geq$ can be negated to make $\leq$ constraints

Another category of linear programming involves where our variables are now exclusively integers, this increases complexity of our problem to NP-H