CO150 - Graphs and Algorithms

Prelude

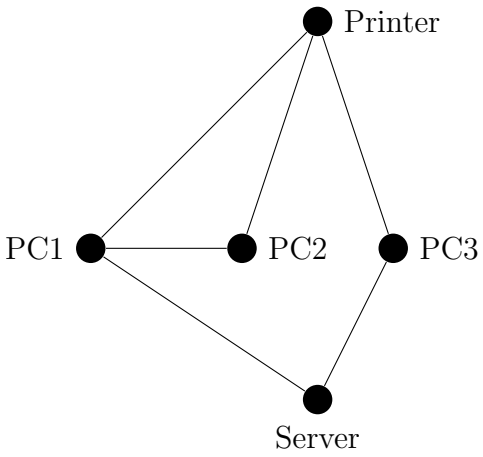
The content discussed here is part of CO150 - Graphs and Algorithms (Computing MEng); taught by Iain Phillips, in Imperial College London during the academic year 2018/19. The notes are written for my personal use, and have no guarantee of being correct (although I hope it is, for my own sake). This should be used in conjunction with the notes.

14th January 2019

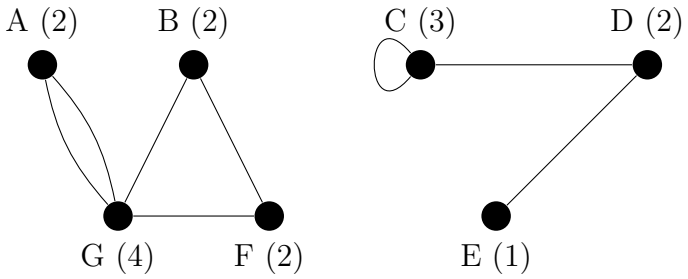
Introduction to the structure of the course;

Part I:	Graphs
Part II:	Graph Algorithms
Part III:	Algorithm Analysis
Part IV:	Introduction to Complexity

An example graph with a real life application;



Note how all the PCs are directly connected to the printer, but PC2 can only reach the server through PC1. On the other hand, we can create a more general graph to display some features that may be less common;



Note that this isn't actually two graphs; it's **disconnected components**. Between A, and G, there are two **parallel arcs / edges**, and C has **loop** with itself. I will continue to refer to this graph as the "example", for the remainder of this section, since it displays properties which we may want to analyse later.

We can say that the left subgraph is robust, as it will remain connected against a single failure. However, the right subgraph isn't robust, as a failure between C, and D, or between E, and D would cause one of the nodes to become disconnected. We can then remedy this by adding a connection between C, and E.

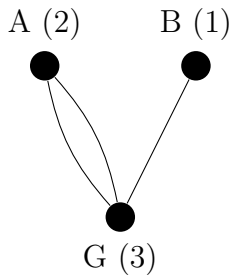
In the graph drawn above, the degrees are also specified - which is the number of arcs connected to it. Note that the degree of C is 3, as we count loops twice for consistency reasons. The sum of the degrees is 16, which is double the number of arcs (8). This is because each arc is counted twice (where it starts, and where it ends), therefore the sum of the degrees is always even. From that, we can then infer that the number of odd nodes (C, and E in our case) must be even. This is trivial to prove with arithmetic.

Subgraphs

We can say that G_1 is a subgraph of G_2 if both of the following criteria apply;

- $\text{nodes}(G_1) \subseteq \text{nodes}(G_2)$
- $\text{arcs}(G_1) \subseteq \text{arcs}(G_2)$

A full (induced) subgraph occurs when we have a set of nodes, X, such that $X \subseteq \text{nodes}(G)$. Every connection between the nodes in X, that was present in G, exists in $G[X]$. Then G' is a full subgraph of G, if $G' = G[X]$ for some X. For example, let $X = \{A, B, G\}$, from the example graph, then we have the following induced subgraph;



If we have some subgraph G', and $\text{nodes}(G') = \text{nodes}(G)$, then it G' spans G.

Adjacency Matrix

For the entry in the matrix $a_{i,j}$, it represents the number of arcs that connect i to j. In an undirected graph, this matrix is symmetric (such that $a^\top = a$). In our example, we're doing the rows, and columns, alphabetically. It's also important to note that we count each loop twice in a diagonal entry. We can determine the degree of a node by taking the sum of its respective row (or column), and find the number of arcs by taking the sum of all the values in the matrix, and then halving it.

$$\begin{array}{c} \\ \text{A} \\ \text{B} \\ \text{C} \\ \text{D} \\ \text{E} \\ \text{F} \\ \text{G} \end{array} \begin{array}{c} \begin{array}{ccccccc} \text{A} & \text{B} & \text{C} & \text{D} & \text{E} & \text{F} & \text{G} \end{array} \\ \begin{bmatrix} 0 & 0 & 0 & 0 & 0 & 0 & 2 \\ 0 & 0 & 0 & 0 & 0 & 1 & 1 \\ 0 & 0 & 2 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 1 \\ 2 & 1 & 0 & 0 & 0 & 1 & 0 \end{bmatrix} \end{array}$$

Adjacency Lists

You'll notice that in our graph, we have a lot of 0s, which makes it less efficient to store as an adjacency matrix; especially if we don't require random access to the degrees. We tend to use n to represent the number of nodes (vertices), and m to represent the number of arcs (edges). You'll note that the size of this is $\leq n + 2m$ (as we have n nodes on the left, and each arc is counted twice, except for loops). Therefore, we can say a graph is sparse if $2m \ll n^2$. Since certain algorithms we work with only look at the arcs incident to a given node, a linked list will be better for sparse graphs.

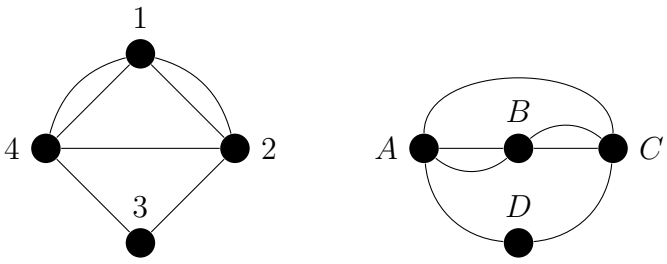
A	→ G, G
B	→ F, G
C	→ C, D
D	→ C, E
E	→ D
F	→ B, G
G	→ A, A, B, F

Big-Oh Notation

I'm too lazy to write out the example, but the idea is that we ignore constant factors, and only consider the most significant term; for example, we could summarise some algorithm that takes $3n^4 + 2n - 4631$ to run as $O(n^4)$. This has significant advantages, since it allows us to abstract away from the implementation / hardware specifics, and instead focus on the factors which determine growth.

Isomorphism

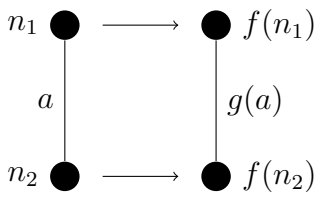
In general, an isomorphism is a bijection that preserves connections. While the two graphs drawn below appear fairly different, they are isomorphic. Mapping from the left, to the right, we know that $3 \mapsto D$, simply because they are the only nodes with degree 2. It's also evident that $1 \mapsto B$, as it's the only node which has two sets of parallel arcs coming out of it. However, it doesn't matter which of 4, or 2, maps to A, or C. Therefore, we can say $4 \mapsto A, 2 \mapsto C$, or $4 \mapsto C, 2 \mapsto A$.



While we're able to check this fairly easily by simply looking at the graph, a computer would have to rearrange the LHS' adjacency matrix to the RHS' (or vice versa).

Given two graphs, G, G', an isomorphism from G to G' consists of two bijections (one-to-one mapping), as well as an additional restriction;

- $f : \text{nodes}(G) \mapsto \text{nodes}(G')$
- $g : \text{arcs}(G) \mapsto \text{arcs}(G')$
- if $a \in \text{arcs}(G)$, with endpoints n_1, n_2, then the endpoints of $g(a)$ are $f(n_1), f(n_2)$ (see the diagram below for a visual example).



In order to confirm whether two graphs are isomorphic, the easiest approach is to first check the obvious; whether the number of arcs, nodes, and loops are the same, as well as the degrees of the nodes. If any of these are different, then the graphs cannot be isomorphic. However if they pass all the tests, then we can attempt to find a bijection on the nodes.

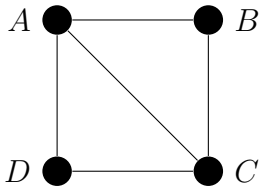
17th January 2019

Complexity

Generally, the process of determining whether two graphs are isomorphic is computationally expensive, hence it has a high complexity. A naive approach would be to check all the permutations, which would then lead to a complexity of $O(n!)$, which is worse than even exponential ($O(2^n)$).

Automorphisms

An automorphism on G is an isomorphism from G to itself. Every graph has at least one automorphism (the identity). Consider the following graph;

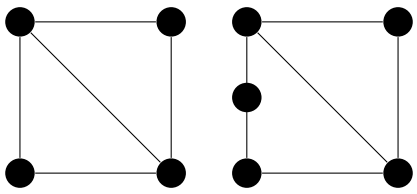


We can do the following method to find the number of automorphisms;

- fix a node, B, it can go to where D is, or stay (2 possibilities)
- take the next node A, it can either stay where it is, or go to where C is (2 possibilities), now fix it
- take the next node C, it can only stay where it is, as it can't go to D since D isn't connected to B, nor does it have a degree of 3 (1 possibility), now fix it
- finally D can only stay where it is (1 possibility)
- multiply all the possibilities, and we have 4 automorphisms

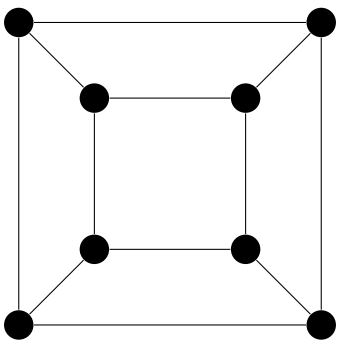
Planar Graphs

We can say a graph is planar if it can be drawn such that no arcs cross. Any non-planar graph contains K_5, or $K_{3,3}$ as a subgraph homeomorphic. We can say that two graphs are **homeomorphic** if they can be obtained by a series of operations such that an arc $x - y$, is replaced by two arcs $x - z$, and $z - y$. For example, the two graphs below are homeomorphic.



There is a linear time algorithm to check whether a graph is planar; however in this case linear time means $O(n + m)$, with the previous definitions.

Any planar graph splits the plane into regions, which are referred to as faces; the graph below splits it into 6 faces (including the outside region). With a graph G that has N nodes, A arcs, and F faces, Euler's formula states $F = A - N + 2$ for any connected planar graph.



Graph Colouring

Any (literal, real-life) map can be converted into a simple planar graph by letting the countries represent nodes, and joining them if they are neighbours. This newly generated graph is known as the dual graph. We can say some graph G is k-colourable, if the nodes of G can be coloured with no more than k colours, therefore every simple planar graph is 4-colourable.

Bipartite Graphs

We can say a graph is bipartite if we can partition nodes(G), into two sets X, and Y, such that no two nodes of X are joined, and likewise for Y. A graph is bipartite $\Leftrightarrow$ it is 2-colourable.

Paths, and Connectedness

A path in a graph is a sequence of adjacent arcs, although normally described by the nodes that we pass through. A path is called **simple** if it doesn't repeat nodes, and a graph is **connected** if there is a path joining any two nodes.

We can define a relation on nodes(G) by $x \sim y \Leftrightarrow$ there is a path from x to y. This is an equivalence relation, as we can prove it's reflexive, symmetric, and transitive.

- $\forall x \in \text{nodes}(G)[x \sim x]$, x is trivially connected to itself, hence it is reflexive
- $\forall x, y \in \text{nodes}(G)[x \sim y \Rightarrow y \sim x]$, as we are working on an undirected graph, this follows trivially
- $\forall x, y, z \in \text{nodes}(G)[x \sim y \wedge y \sim z \rightarrow x \sim z]$, follows trivially by definition of paths

A cycle (circuit) is a special type of path that finishes where it starts, has at least one arc, and doesn't reuse an arc. A graph which doesn't have cycles is **acyclic**.

21st January 2019

Euler Paths / Circuits

An Euler path is a special type of path where each arc is used exactly once, and an Euler circuit is a cycle which uses each arc exactly once (therefore an EC is an EP which finishes at the start node). A connected graph has an EP $\Leftrightarrow$ there are 0, or 2 odd nodes, and there is an EC $\Leftrightarrow$ there are no odd nodes.

We can justify it by saying that any intermediate node (ones which aren't the start node) have to be entered, and exited the same number of times (otherwise it wouldn't be an intermediate) node. Therefore, if 2 nodes of odd degree, then it follows that we start from one, and end on the other.

Consider the following nodes; n, n' being the start, and end (the odd nodes of the path), and arbitrary intermediate nodes i. Start at n, and keep going until we can go no further (n'). If we've stopped at n, then there must be a spare arc, as we've started, and 'ended' at an odd node. If we stop at some arbitrary $i \neq n'$, then we've still got more arcs, since i is even.

Hamiltonian Path / Circuits

A Hamiltonian path is one that visits each node exactly once, and similarly a Hamiltonian circuit returns to the start node. For this, we will only consider simple graphs, since we won't ever follow a loop, or a parallel arc. In order for there to be a HP, we need a connected graph, and for a HC to exist, each node must have a degree ≥ 2. To determine whether a circuit exists, we can take a brute force approach, since a circuit is really just a permutation on the set of nodes; such that for n nodes, we have $\pi : \{1, ..., n\} \mapsto \{1, ..., n\}$. However, for this to be a circuit, we need $\pi(i)$ to be adjacent to $\pi(i+1)$, and so on. As we have $n!$ possible circuits, this is far too slow. There exists a dynamic programming approach that reduces this to $O(n^2 2^n)$, but that is still exponential. Compared to EPP, which has $O(n^2)$ time. HCP has been shown to be NP-complete, and are therefore not solvable in polynomial time.

Trees

A tree is an acyclic connected graph (whether we specify it's rooted, or non-rooted, depends on the author). The root of G is a distinguished node. Assuming a rooted graph, the depth of a node x is the distance along the unique path from the root to x. If x isn't the root node, the parent of x is the node directly before it in the path from the root to x. The depth of tree is the maximum of the depths of all its nodes.

A spanning tree of a graph G, is a tree, T, such that nodes(T) = nodes(G), and arcs$(T) \subseteq$ arcs(G). The spanning trees are not necessarily unique.

Directed Graphs

While we generally cover undirected graphs in this course, it makes sense in some applications for the arcs to be directed. For each $a \in$ arcs(G), it is associated with an **ordered** pair of nodes. In diagrams, these are shown with arrows. In a path for $a_1, ..., a_n$, the source of a_{i+1}, must match the target of a_i. We define the indegree as the number of arcs entering, and likewise the outdegree is the number of arcs leaving. For any directed graph, the sum of the indegree of all nodes, and the outdegree of all nodes is equal to the number of arcs. We say a directed graph is strongly connected if there exists a path between any two nodes in G.

24th January 2019

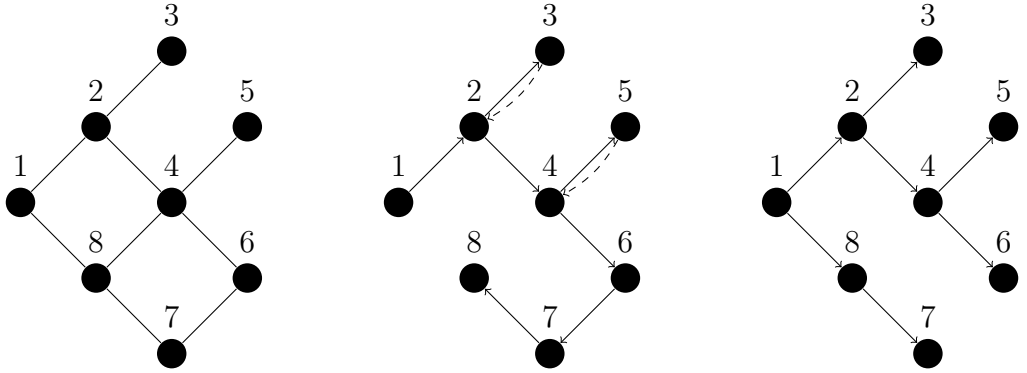
Tree Traversal Algorithms

The two types of traversal covered in this course are depth-first search (DFS), and breadth-first search (BFS). While they are similar in many ways, they also have quite a few differences.

In depth first search, we choose one of the adjacent nodes to the start; from there, we then spawn another depth first search. This is done recursively until there aren't any more (unvisited) adjacent nodes. At this point, the spawned DFS returns back to the parent node, where it checks the next adjacent node (normally ordered by how the adjacency list / matrix is stored). It does this until we have visited all of the nodes, and then returns back all the way to the start.

In contrast to DFS, breadth-first search goes through all the adjacent nodes, and then goes deeper. This means that we only check the nodes adjacent to the ones adjacent to the start node, after all of the nodes directly adjacent to the start node have been visited.

You'll note that the distance between 1, and 8, is 5 in DFS, but only 1 in BFS. In BFS, the depth of any node is its distance from the start. However, both generate spanning trees on G.



full graph DFS BFS

In order to formalise this, let us consider the graph to be traversed as an adjacency list, a boolean array of nodes (which are visited), and the parent is the parent node in the search tree. The output will be the nodes visited in order.

```
procedure dfs(x):
  visited[x] = true
  print x
  for y in adj[x]:
    if not visited[y]:
      parent[y] = x;
      dfs(y)
      # at this point, control is returned to x
      # we don't need the parent in this case, but other applications may use it
```

The running time of DFS is $O(n+m)$, therefore it's linear. However, this implementation may have some overhead due to recursion.

```
procedure bfs(x):
  visited[x] = true
  print[x]
  enqueue(x, Q)
  while not isEmpty(Q):
    y = front(Q)
    for z in adj[y]:
      if not visited[z]:
        visited[z] = true
        print z
        parent[z] = y
        enqueue(z, Q)
    dequeue(Q)
```

The size of the queue represents the breadth of the front. Once again, the complexity is $O(n+m)$.

Applications of Traversal

The algorithms used above also work on non-connected graphs. If we analyse the set of visited nodes, and see that it isn't the same as the set of all nodes, then it is clear that the graph is not connected. As such, we have an $O(n+m)$ algorithm for detecting non-connected graphs.

We can trivially say that a graph has a cycle if it has $\geq n$ arcs. Alternatively, we can use DFS; if we encounter a node that has already been visited (other than by backtracking), then it has a cycle.

It's also trivial to modify BFS to find the distance of each node, by having a running counter. Due to how BFS is implemented, we can also extract the shortest path from `y`; as `y, parent[y], parent[parent[y]], ..., start`.

Weighted Graphs

We can associate a cost with each arc on a network. We can define a weighted graph as a graph, G, with a weight function $W : \text{arcs}(G) \mapsto \mathbb{R}^+$. With weights, we're able to consider the following problems; finding an MST (minimum spanning tree), finding shortest paths, and finding a shortest circuit.

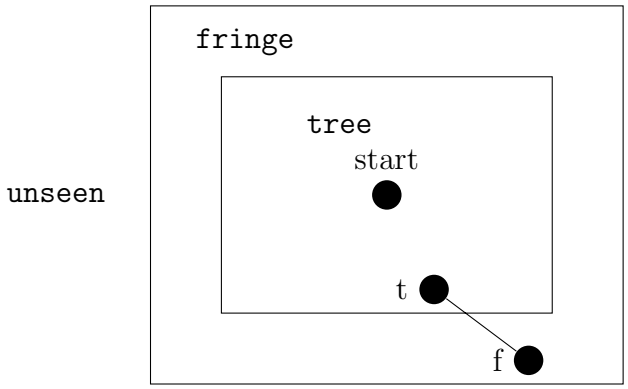
We're only going to consider simple graphs, as there is no point in taking a loop if we're trying to minimise cost, nor is there any point in taking the more expensive arc in a parallel arc.

Prim's Algorithm

We can say that T is a minimum spanning tree for Gm if T is a spanning tree for G, and no other spanning tree for G has a smaller weight. Once again, MSTs do not have to be unique.

The main idea in Prim's algorithm is to add the shortest arc that will extend the tree. This is an example of a greedy algorithm, which gives a short-term advantage but may not be the best overall (it

is in this case). At any stage in Prim's algorithm, we have three types of nodes; tree nodes (which are already part of the MST), candidate nodes - which are fringe nodes adjacent to a tree node, and the rest are unseen nodes. At the start, all nodes are unseen.



The general idea is to pick a random node (doesn't matter which, as all nodes will be in the MST by definition). This is now the start node, and therefore part of the MST (hence classified as `tree`). Now classify all the nodes adjacent to this node as `fringe`. While the fringe isn't empty, select the arc with minimum length between a tree node t, and a fringe node f. Classify f as `tree`, and add the arc (t, f) to the tree. Reclassify all the unseen nodes adjacent to f as `fringe`.

The while loop is executed roughly n times, as it will be executed for each node. However, in the worst case, when we're selecting a minimum arc, that takes $n + m$. Therefore, we can calculate this to be an $O(n(n+m))$ algorithm. This is a more naive approach, we can improve this by choosing candidate arcs; as we're avoiding redoing work. If we consider a parent function, such that the parent of f (`fringe`) is t (`tree`), such that the arc (t, f) has the least weight. This can be summarised in two parts; first the initialisation, and then the execution of the algorithm;

```
1  tree[start] = true
2  for x in adj[start]:
3    fringe[x] = true
4    parent[x] = start
5    weight[x] = W(start, x)
6  while not isEmpty(fringe):
7    select f such that weight[f] is minimum
8    fringe[f] = false
9    tree[f] = true
10   for y in adj[f]:
11     if not tree[y]:
12       if fringe[y]:
13         # updating arcs if we can get a lower weight
14         if W(f, y) < weight[y]:
15           weight[y] = W(f, y)
16           parent[y] = f
17       else:
18         # we haven't seen y yet
19         # this can probably be shortened with the above
20         fringe[y] = true
21         weight[y] = W(f, y)
22         parent[y] = f
```

We're still iterating through the loop n times. During the while loop, we're checking whether the fringe is empty (an $O(n)$ operation), finding the minimum fringe (also an $O(n)$ operation), and updating the candidate arc (if needed), which is constant time. Therefore the inner loop has a complexity of $O(n)$, hence the overall algorithm has a complexity of $O(n^2)$. In the worst case, m can be as large as n^2, which would be problematic in the first implementation.

However, we have a greedy algorithm, and therefore need to prove its correctness. Let us represent the trees constructed at each iteration as $T_0, T_1, ..., T_k, T_{k+1}...$; where T_0 is juts the initial start node, and you get T_{k+1} from T_k, by adding some arc a_{k+1}. Hence it follows that T_k has $k+1$ nodes (by induction, probably). With this algorithm, we will end up with n nodes, by definition of a spanning tree, therefore we end up returning T_{n-1}.

The goal here is to now show each T_k is a subgraph of an MST T' of G. Trivially, in the base case T_0, it has one node, and no arcs, therefore $T_0 \subseteq T'$.

Assume that $T_k \subseteq T'$, where T' is an MST of G. Let there be some node in the tree x, a node in the fringe y, and an arc joining them a_{k+1}. We can now consider both cases; if $a_{k+1} \in \text{arcs}(T')$, then $T_{k+1} \subseteq T'$, and is trivial. However, suppose that $a_{k+1} \notin \text{arcs}(T')$, there still has to be a path in T' from x, to y (by properties of a spanning tree). Therefore, we can form a cycle through some other tree node x', to a fringe node y', through some arc a (such that we can connect y, and y'). As Prim's chose a_{k+1}, instead of a, there we can deduce that $W(a_{k+1}) \leq W(k)$, and therefore $W(T'') \leq W(T'))$. However, since all MSTs have the same weight by definition, we can say that $W(T'') = W(T'))$. Also, $T_{k+1} \subseteq T''$, therefore the induction step is complete.

We can also implement Prim's with a priority queue. The PQ requires us to have some key of x, for each item x. In this case, we will likely use the weight of the candidate arc. We have the following operations on PQ;

- `Q = PQCreate()`
- `isEmpty(Q)`
- `insert(Q, x)`
- `getMin(Q)`
- `deleteMin(Q)`
- `decreaseKey(Q, x, newkey)`

The implementation is as follows;

```
1  Q = PQCreate()
2  for x in nodes(G):
3    key[x] = ∞ # some arbitrary large number
4    parent[x] = null
5    insert(Q, x)
6  decreaseKey(Q, start, 0)
7  while not isEmpty(Q):
8    f = getMin(Q)
9    deleteMin(Q)
10   tree[f] = true
11   for y in adj[f]:
12     if not tree[y]: # therefore in Q
13       if W(f, y) < key[y]:
14         decreaseKey(Q, y, W(f, y))
15         parent[y] = f
```

With a priority queue of length N, all operations have complexity $\log(N)$ (other than `isEmpty`, and `getMin`, which are constant time), with a good implementation. Overall, we have a complexity of $O(m\log(n))$, given that $n < m$. In a dense graph, classic Prim is better.

28th January 2019

Kruskal's Algorithm

An even greedier approach is to take the shortest arc that hasn't been included in the tree, except for ones that would cause a cycle. In intermediate stages, we have a forest (which is an acyclic graph),

and not a tree (a connected acyclic graph).

In our implementation, we need to do two things; look at each arc in ascending order (we can either sort the arcs right at the start, or use a priority queue). We will also need to use **dynamic equivalence classes**, which prevents adding arcs that would cause cycles. We can generate equivalence classes if they belong to the same connected tree, and an arc (x, y) can only be added if x, and y are in different equivalence classes. If the arc is added, the classes are merged.

We can use the **Union-Find** data type to implement these DECs. Let each set have a leader element, which represents the set. We can **find** the leader of the equivalence class, and union (merge) two classes (discussion of the implementation is in the next section);

- `sets = UFCreate(n)` creates a family of singleton sets, with `find(sets, x) = x`
- `x' = find(sets, x)` finds the leader `x'` of `x` within `sets`
- `union(sets, x, y)` merges the sets led by `x`, and `y`, and choose one to be the leader

Consider G with n nodes numbered $[1, n]$;

```
1  Q = PQCreate() # arcs of G with the weights as keys
2  sets = UFCreate(n)
3  F = ∅
4  while not isEmpty(Q):
5    (x, y) = getMin(Q)
6    deleteMin(Q)
7    x' = find(sets, x)
8    y' = find(sets, y)
9    if x' != y':
10     add (x, y) to F
11     union(sets, x', y')
```

With a weighted union (non-binary tree) implementation of Union-Find, we have a complexity of find being $O(\log(n))$, and a union of $O(1)$;

- $O(m)$ inserts for the PQ, which takes $O(m\log(m))$
- $O(m)$ `getMins`, `deleteMins`, which both take $O(m\log(m))$
- $O(m)$ `finds`, which takes $O(m\log(n)$
- $O(n)$ `unions`, which takes $O(n)$

Assuming $m \geq n$, as it's normally the case, the overall time is $O(m\log(m))$. However, we know that the number of arcs in a simple graph is bounded by n^2, hence the complexity is $O(m\log(n))$ which is the same as a PQ implementation of Prim's.

31st January 2019

Union-Find Implementations

A naive implementation for the union-find is to store an array of leader nodes, where it initially stores itself as its leader. Finding is $O(1)$, but union will be $O(n)$, which means our implementation of Kruskal's will be $O(n^2)$.

Instead of taking the naive approach, we can consider it as a non-binary tree, with the root node being the leader for the equivalence class. When two trees are merged, with leaders `x`, and `y`, we either set `parent[x] = y`, or vice versa. This now makes a merging operation $O(1)$, which is much better. However, finding the parent will now involve traversing up the `parent` chain, which would be $O(n)$ in the worst case (on a tree where it's basically a linked list). This still keeps Kruskal's at $O(n^2)$, which isn't an improvement.

The goal for us now is to minimise the depth of the tree, by taking a weighted union instead of some arbitrary choice. The rule for a weighted union is to append the tree of lower size (number of nodes, not depth), to the one of greater size. Storing, and updating, size is a very simple operation. By the weighted union, the depth of a tree of size k is $\leq \lfloor \log(k) \rfloor$.

Path Compression

We can reduce the union-find complexity for Kruskal's to $O((n+m)\log^*(n))$, where $\log^*(n)$ is an extremely slow-growing function, such that it's ≤ 5 for any n we might use. The idea is to set the parent of some node we're finding by the root node as follows;

```
1 procedure cfind(x):
2   y = parent[x]
3   if y == x:
4     root = x
5   else:
6     root = cfind(y)
7     if root != y: # path compression
8       parent[x] = root # path compression
9   return root
```

Comparison

We can refer to the table below to summarise the complexities of each of the algorithms we're referring to. On dense graphs, where m has order n^2 (is large), then Kruskal's gives a complexity of $O(n^2\log(n))$, therefore classic Prim's is better. ON the other hand, where m is small, around the order of $n\log(n)$, then Kruskal's or PQ Prim's is better, as we have $O(n\log^2(n))$.

algorithm	complexity
Kruskal	$O(m\log(n))$
Prim with binary heap PQ	$O(m\log(n))$
Classic Prim	$O(n^2)$

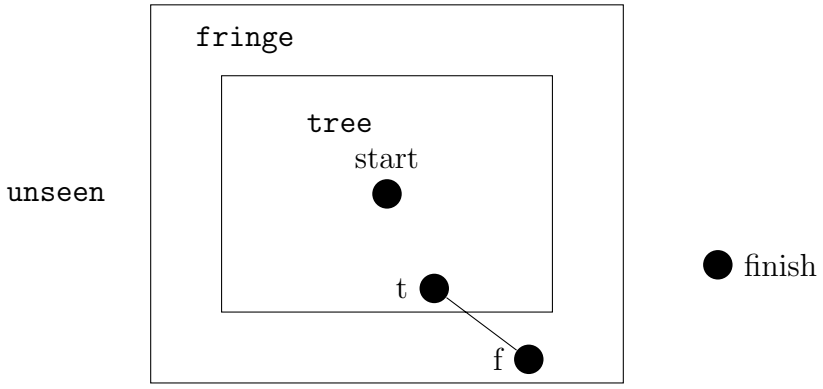
Fibonacci Heaps

If we implemented a PQ with a Fibonacci heap instead of a binary heap, we have all constant time operations (other than `deleteMin`, which would have a complexity of $O(\log(n))$. The complexity of Prim's with a Fibonacci heap PQ is $O(m + n\log(n))$, however the memory usage, as well as constant factors, can be higher, therefore a binary heap might still be preferred.

Shortest Path Problem

Given we have a weighted graph G, and nodes S, and F, representing start, and finish respectively, we have an $O(n^2)$ algorithm to find the shortest path of a single pair developed by Dijkstra. There is also an algorithm by Floyd, which finds all the shortest paths between any pairs, in $O(n^3)$ time.

As Dijkstra's is closely related to Prim's, we can use the same diagram from before, with some modifications (not just because I'm too lazy to draw another one with TikZ). Once again, the nodes are classified into `tree`, `fringe`, and `unseen`. However, we modify it by saying that we've already computed the shortest path from the start node, to all the `tree` nodes - the path given by the tree. For the `fringe` nodes, we know the shortest path using the tree, although this path can be improved as the tree grows (similar to the inductive step in the proof for Prim's). However, instead of choosing the shortest candidate arc, we choose the shortest distance from the staff.



The implementation fro this is similar to Prim's algorithm, but we also need to store its parent node, and the distance of the shortest path known. At any stage, the next node to be added is the fringe node with the smallest distance from the start. The fringe is then updated, possibly improving the shortest path. Once again, we can keep checking parents to obtain the path we've taken.

```
tree[start] = true
for x in adj[start]:
  fringe[x] = true
  parent[x] = start
  distance[x] = W(start, x)
while not tree[finish] and not isEmpty(fringe):
    select f where distance[f] is minimum
    fringe[f] = false
    tree[f] = true
    for y in adj[f]:
      if not tree[y]:
        if fringe[y]:
          if distance[f] + W(f, y) < distance[y]:
            # updating distance if we have a better option
            distance[y] = distance[f] + W(f, y)
            parent[y] = f
        else:
          fringe[y] = true
          distance[y] = distance[f] + W(f, y)
          parent[y] = f
return distance[finish]
```

As it's merely a light modification on Prim's, we can easily justify that the running time is $O(n^2)$. We can prove termination, as we're increasing the tree each iteration. Once again, we can implement this with a PQ as such;

```
Q = PQCreate()
for x in nodes(G):
  key[x] = ∞
  parent[x] = null
  insert(Q, x)
decreaseKey(Q, start, 0)
while not tree[finish] and not isEmpty(Q):
  f = getMin(Q)
  deleteMin(Q)
  tree[f] = true
  for y in adj[f]:
    if not tree[y]:
      if key[f] + W(f, y) < key[y]:
```

```
14          decreaseKey(Q, y, key[f] + W(f, y))
15          parent[y] = f
```

Once again, the complexity is the same, with either the binary heap, or the Fibonacci heap implementation.

4th February 2019

A* Algorithm

Dijkstra's algorithm omits any sense of direction, and expands outwards until we reach the finish node. A* modifies this by allowing for some heuristic function $h(x)$ by underestimating the distance from any node to the finish node. On a map, we could say that h represents the Euclidean distance between any two cities.

We can calculate the cost of going to a node as $F(x) = g(x) + h(x)$, where $g(x)$ is what we defined as the distance between the target node, and the start node. Given a start node, we calculate the 'cost' of the node by taking the F values of them, and then we select the smallest one. We then add that node, x, to the tree. At this point, we can then consider the nodes adjacent to x, and calculate their costs, by using the same method. This checks the smallest cost nodes, that aren't already checked, but we will end up saving time since the heuristic function adds a bias towards the nodes that are heading in the direction of the finish. In order for a heuristic function to be considered consistent, it needs to fulfil the following criteria;

- for any two adjacent nodes, x, y, we have $h(x) \leq W(w, y) + h(y)$
- $h(\texttt{finish}) = 0$

Consistency is a stronger property than a function being admissible (if $h(x) \leq$ the weight of the shortest path from x to the finish). We can solve this trivially with the consistency criteria, by setting $y = \texttt{finish}$, as we'd then have $h(x) \leq W(x, \texttt{finish})$. The set of tree nodes is often referred to as the **closed set**, and the set of fringe is the **open** set. If we have a heuristic $h(x) = 0$, albeit useless, then we get Dijkstra's algorithm. Thus we can say that the complexity of A* is the same as Dijkstra, in the worst case. We can justify correctness similar to Dijkstra's. Our 'invariant' is that if the node x is a `tree`, or `fringe` node, and not the `start`, then `parent[x]` is a tree node. If x is a tree node, then $g(x)$ is the length of the shortest path (once again, not including `start`). However, if x is a fringe node, then $g(x)$ is the length of the shortest path, where all nodes except x are tree nodes.

Let us assume that we have a different, shorter path P, which may exist outside the tree nodes. Then it follows that $\text{len}(P) < g(x)$. Let there be some y which is the first node in that isn't part of the tree to be on P, and P_1 be the path from `start` to y, and P_2 be the path from y to x. We know that $F(y) = g(y) + h(y)$, by definition. Therefore $\leq g(y) + \text{len}(P_2) + h(x)$, by the consistent property of h. And also that $\leq \text{len}(P_1) + \text{len}(P_2) + h(x) = \text{len}(P) + h(x)$. As we've assumed that it has a shorter path, it follow that it must be $< g(x) + h(x) = F(x)$. Therefore $F(y) < F(x)$, but that contradicts our choice of x, since we chose the lowest F.

For the sake of completeness; this is the algorithm implemented with a PQ;

```
1  Q = PQCreate()
2  for x in nodes(G):
3    g[x] = ∞
4    key[x] = ∞
5    parent[x] = null
6    insert(Q, x)
7  g[start] = 0
8  decreaseKey(Q, start, g[start] + h[start])
9  while not tree[finish] and not isEmpty(Q):
10   x = getMin(Q)
```

```
    deleteMin(Q)
    tree[x] = true
    for y in adj[x]:
      if not tree[y]:
        if g[x] + W(x, y) < g[y]:
          g[y] = g[x] + W(x, y)
          decreaseKey(Q, y, g[y] + h[y])
          parent[y] = x
```

7th February 2019

Transitive Closure

Let there be some binary relation $R \subseteq X^2$. We can use the definition of transitive closure from **CO142** (whoops, probably should've done Discrete Structures before trying to learn this module), as $R^+ = \bigcup_{\infty}^{k=1} R^k$.

However, instead of using ∞, we can use $|X|$ instead, given that X is a finite set. We can interpret R as a directed graph on G, where the ordered pairs of R are arcs in G, and elements in X are nodes in G. Therefore, we can deduce that $R^k(x, y) \Leftrightarrow$ there is a path of length k, from x to y. Therefore $R^+(x, y) \Leftrightarrow$ there is a path of length ≥ 1 from x to y.

Assuming we have the set $X = \{1, ..., n\}$, and $A[i, j] = 1$ if $R(i, j)$, otherwise 0, then we have the adjacency matrix A of G. Computing R^k is then trivial with matrix multiplication, such that $R^k(i, j) \Leftrightarrow A^k[i, j] > 0$.

Then we can simply calculate $B = \sum_{k=1}^{n} A^k$, which means that $R^+(i, j) \Leftrightarrow B[i, j] > 0$

Warshall's Algorithm

Suppose we have nodes $\{1, ..., n\}$. Considering a path $p = x_1, x_2, ..., x_k$ from x_1 to x_k, we can say that x_m is an intermediate nodes, $\forall m \in [2, k-1]$. We want to look for paths that use nodes $\leq i$ as intermediate nodes. Defining a matrix $B_k[i, j] = 1 \Leftrightarrow$ if there exists a path from i to j that uses only intermediate nodes $\leq k$ (otherwise the value in the matrix is 0).

Therefore, $B_0[i, j] = A[i, j]$ as we only consider paths of length one, since there are no intermediate nodes ≤ 0. Since B_n allows all possible intermediate nodes, we have all possible paths, therefore $R^+(i, j) \Leftrightarrow B_n[i, j]$.

Consider how to compute B_k from B_{k-1}. Suppose we have some path P from i to j, using intermediate nodes $\leq k$. The path either contains k as an intermediate node, or it doesn't. In the former, k is not an intermediate node, therefore $B_{k-1}[i, j]$. However, if it is, we only consider when k occurs once (if it repeats, then it's not helpful - we just remove the path between the two ks). Then we have some path i to k, and another path k to j, therefore it follows that $B_{k-1}[i, k]$, and also $B_{k-1}[k, j]$. This has a trivial complexity of $O(n^3)$, which is much more efficient than the matrix method for closure.

```
input A
copy A into B # B = B_0
for k = 1 to n:
  # B = B_k - 1
  for i = 1 to n:
    for j = 1 to n:
        B[i, j] = B[i, j] or (B[i, k] and B[k, j])
  # B = B_k
#B = B_n
return B
```

Floyd's Algorithm

Returning to the all pairs shortest path problem on some weighted directed graph G. This can be solved with a modification of Warshall's. Once again, G has nodes $\{1, ..., n\}$, and an adjacency matrix A. We now consider $B_k[i, j]$ to be the length of the shortest path between i, and j which has intermediate nodes $\leq k$. If there isn't a path, then let the value be ∞. Once again, like with Warshall's, we let $B_n[i, j]$ represent the length of the shortest path between i, and j. Assume that we know B_{k-1}, and we want to get B_k. Suppose there exists some shortest path P from i to j, using intermediate nodes $\leq k$, with $\text{len}(P) = d$. Once again, if we consider the two cases, the case where k is not in P, is trivial, and the second where k is in the path, only occurring once. We can then calculate the length between the two as the minimum of $B_{k-1}[i, j]$, and $B_{k-1}[i, k] + B_{k-1}[k, j]$. If there aren't any shortest paths, then we still get ∞. Once again, it has the same complexity of $O(n^3)$.

```
input A
(for all i, j <= n) B[i, j] = 0       if i = j
                              A[i, j] if i ≠ j and there is an arc (i, j)
                              ∞       otherwise
for k = 1 to n:
  for i = 1 to n:
    for j = 1 to n:
        B[i, j] = min(B[i, j], B[i, k] + B[k, j])
return B
```

Dynamic Programming

The previous two algorithms are examples of dynamic programming. In dynamic programming, we break down the problem into sub-problems, and the smaller sub-problems are ordered to culminate in the main problem. We move through the smaller sub-problems in order, and solve each sub-problem with the stored results of the previous one (and then storing this result for use by the next one). The main problem is solved as if it were the final sub-problem (deep really).

Travelling Salesman Problem

Given some complete weighted graph (G, W), we want to find a way to tour the graph visiting each node exactly once, and travelling the shortest possible distance. We consider complete graphs, as we can construct an 'arc' with an extremely high weight.

This problem is related to the Hamiltonian Circuit Problem, and the Shortest Path Problem. TSP is NP-complete, and unlikely have a polynomial solution. An improvement on the factorial approach is the Bellman-Held-Karp algorithm (will be written as BHK), which has a running time of $O(n^2 2^n)$.

Once again, we have (G, W) with nodes $\{1, ..., n\}$. We can fix a start node, say 1 as it doesn't matter where you begin, since the problem would require you to visit all of them anyways. For each $x \neq 1$, and each $S \subseteq \text{nodes} \backslash \{1, x\}$, we can find and store the minimum cost of $C(S, x)$ from node 1 to x, using the set of intermediate nodes S (you have to use exactly the set of intermediate nodes).

We can use this to calculate the solution to TSP as $\min(C(\text{nodes} \backslash \{1, x\}, x)) + W(x, 1)$, by trying each $x \neq 1$. This is done by calculating all $C(S, x)$ in increasing size order, from 0 to $n - 2$ (which is all nodes excluding 1, and x). Clearly, with the base case $C(\varnothing, x) = W(1, x)$, as we aren't allowed any intermediate nodes. If we assume that we know $C(S, x)$ for all S of size k. Now, consider a set S', with a size of $k + 1$, such that $S' = S \cup \{y\}$, where y is some last node (and $S = S' \backslash \{y\}$), Therefore, it follows that the cost of a path from 1 to x with intermediate nodes S' is $C(S, y) + W(y, x)$, therefore it follows that $C(S', x) = min(C(S, y)) + W(y, x)$.

```
input (G, W)
choose start ∈ nodes(G)
for x in nodes \ {start}:
```

```
    c[∅, x] = W(start, x)
  # process S in size order
  for S ⊆ nodes \ {start} with S ≠ ∅
    for x in nodes \ (S ∪ {start}):
    # find c[S, x]
    c[S, x] = ∞
    for y in S:
      c[S, x] = min(c[S \ {y}, y] + W(y, x), c[S, x])
  opt = ∞
  for x in nodes \ {start}:
    opt = min(c[nodes \ {start, x}, x] + W(x, start), opt)
  return opt
```

For each subset of `nodes`, we do $O(n^2)$ the two for loops, with an overall complexity of $O(n^2 2^n)$.

11th February 2019

Approximate Methods for TSP

We could easily try a nearest neighbour heuristic for TSP, such that we choose the shortest available arc that doesn't violate any rules. This can fail though, since we may fall into a point where we're forced to use an extremely high weight arc.

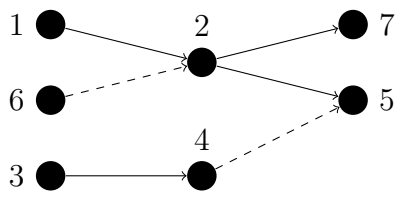
Topological Sorting

Suppose there is a list of tasks, with prerequisites (therefore some have to be performed others, and these may allow others to be performed). This is seen as an acyclic, directed graph (if it were cyclic, then there would be a sort of circular dependency). Find a total ordering, such that for nodes $x_1, ..., x_n$, for any $i, j \leq n$, if $j > i$, there is no path from x_j to x_i.

Given a DAG G, let $x \leq y \Leftrightarrow$ there is a path from x to y, then $\subseteq$ is a (weak) partial ordering, as it's reflexive, transitive, and antisymmetric. If you have a partial ordering $(X, \leq)$, then we can create an acyclic directed graph with nodes X, and arcs $\{(x, y) : x \leq y\}$.

We can perform a topological sorting by using DFS, as it inherently finishes with all the nodes reachable from x before we finish with x. We can add a node to the end of a sorted list, once we've finished processing it (by exiting the node). In the example below, our visiting order is 1, 2, 5, 6, 3, 4, 6. However, we want to consider the order we exit them, not the order in which we enter. Thus we get the exit order 5, 7, 2, 1, 4, 3, 6. Therefore we end up with the sort 6, 3, 4, 1, 2, 7, 5.

```
procedure dfsts(x):
  entered[x] = true
  for y in adj[x]:
    if entered[y]:
      if not exited[y]:
        ERROR # cycle
    else:
      parent[y] = x
      dfts(y)
exited[x] = true
ts[index] = x
index = index -1
```



Algorithm Analysis

Given we have some problem P, and S for all the possible solutions for P. In this course,we only consider the complexity. A harder, more theoretical question, would be "can we improve our existing algorithm, or have we found the best algorithm for P?". In order to reason about this, we need to consider what is the least amount of work every member of S must to do solve P - this gives us a lower bound complexity on P. This is important as we're not just considering a large amount of solutions, but all possible solutions that exist.

For this, we will examine searching, and sorting algorithms; because these are frequently used (therefore efficiency is very important), and analysis for them have already been carried out (such that we know the optimal algorithms). If we can prove an algorithm has the complexity exactly on the lower bound, then we've proven that it's the best possible algorithm.

Unordered List

Consider a list of elements (that allows for random access) L, and some target element x. We can consider a simple, linear approach; inspect each element, until the end of the list. For this, we have a best case of 1 (where it's the first item we check), and a worst case of n (where the item is either at the end of the list, or not in the list at all). Note that here we are counting comparisons.

In worst case analysis, we denote it with $W(n)$, which is the largest number of comparisons for an input size n. We could also consider the average case, which is $A(n)$, however this is much harder to analyse as we might need to know (or make an assumption) about its probability distribution.

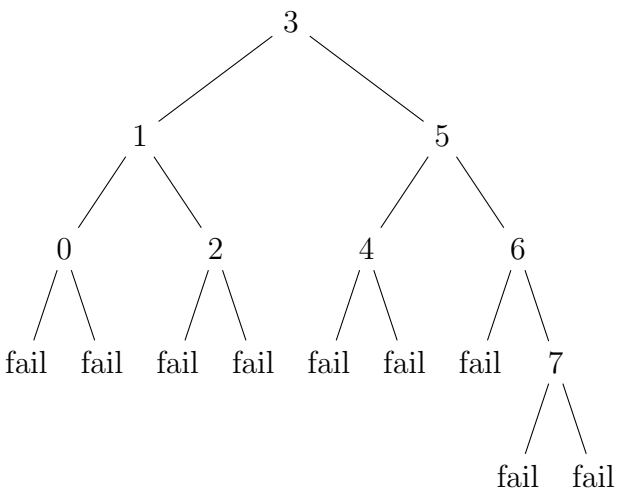
Here, let us reason about how linear search is optimal for an unordered list. Take any A which solves the search problem. We can claim that if A says the item isn't found, it must've inspected every element. In order to prove this by contradiction, suppose that A didn't inspect $L[k]$, which is an arbitrary index in the list. Now place some item, x into $L[k]$, and A would then return that the item wasn't found, which is incorrect. Therefore, in the worst case, n comparisons are needed.

14th February 2019

Ordered List

If the problem were now changed to be searching an ordered list, linear search would still be applicable, but no longer optimal.

We can now consider binary search, which divides the list (and we're able to do that due to the ordered nature of the list). It can either be represented by pseudocode (which is trivial to work out), or as the following decision tree. Note that for sake of time, and because I'm lazy, if it goes down the right left hand side, for some node n, it means that $x < L[n]$, and for the right hand side, it would mean that $x > L[n]$. If $x = L[n]$, then we've solved it (note that the leaf nodes represent my result in this exam);



We can say that the worst case, $W(8) = 4$, as it's the depth of the decision tree. However, this is a big improvement on linear search, as we are reducing the worst case to a logarithm of n, where the difference would be much more apparent in larger data sets. Since the worst case of binary search is the depth of the tree, we can say that it's the optimal solution.

While we can fairly easily see the worst case by looking at the decision tree, it's not as apparent for other algorithms, or larger sets of data. We can use a recurrence relation to solve it;

$$W(1) = 1$$
$$W(n) = 1 + W(\lfloor \frac{n}{2} \rfloor)$$

We can solve this by repeated expansion;

$$\begin{aligned} W(n) &= 1 + W(\lfloor \frac{n}{2} \rfloor) \\ &= 1 + [1 + W(\lfloor \frac{n}{4} \rfloor)] \\ &= 1 + 1 + 1 + W(\lfloor \frac{n}{8} \rfloor)] \\ &\quad ... \\ &= 1 + ... + 1 + W(1) \end{aligned}$$

Now, we can say that the number of 1s is the number of times the number is divisible by 2, hence k where $2^k \leq n < 2^{k+1}$. Therefore $k = \lfloor \log_2(n) \rfloor$, Hence it follows that $W(n) = 1 + \lfloor \log_2(n) \rfloor$.

We can represent search algorithm, considering comparisons, as a binary tree - we either have the item being less than, greater than, or equal. In the latter, we can just terminate the algorithm. We can propose that if a binary tree has depth d, then it has $\leq 2^{d+1} - 1$ nodes. This can be proven by induction over natural numbers.

Given that the worst case performance of some search algorithm A is $d + 1$ (where d is the depth, and we can use the inequality for the nodes above). It follows that $d + 1 \geq \log(n + 1)$ by taking logs of both sides. And this can be proven to be $W(n)$.

Orders

Suppose we have some algorithm $W(n) = 7n^3 + 1000n + 32$, as n gets larger, $7n^2$ is the most important term. If we ignore the constant, $W(n)$ therefore has order n^2. We prefer algorithms of lower order, even if a higher order algorithm may have a lower operation time for small n. The constant factor can also be ignored.

The main types of orders we encounter are; polynomial, exponential, and logarithmic. The hierarchy for logarithmic (with polynomial) is as follows;

$$1 \quad \log(n) \quad n\log(n) \quad n^2 \quad n^2\log(n)$$

Let two functions $f, g : \mathbb{N} \mapsto \mathbb{R}^+$;

- f is $O(g) \leftrightarrow \exists m \in \mathbb{N} \exists c \in \mathbb{R}^+ [\forall n \geq m [f(n) \leq c \cdot g(n)]]$
- f is $\Theta(g) \leftrightarrow f$ is $O(g)$, and g is $O(f)$

  the Θ means order

We know that matrix multiplication is a $O(n^3)$ problem, but there exists a lower bound of $\Theta(n^2)$.

18th February 2019

Strassen's Algorithm

Assume that we are trying to multiply two $n \times n$ matrices; $AB = C$. Start with $n = 2$, then it follows that

$$C = \begin{bmatrix} a_{1,1}b_{1,1} + a_{1,2}b_{2,1} & a_{1,1}b_{1,2} + a_{1,2}b_{2,1} \\ a_{2,1}b_{1,1} + a_{2,2}b_{2,1} & a_{2,1}b_{1,2} + a_{2,2}b_{2,2} \end{bmatrix}$$

This algorithm takes 8 multiplications, and 4 additions. However, Strassen's algorithm does $n = 2$ in 7 multiplications (and 18 additions);

$$C = \begin{bmatrix} x_1 + x_4 - x_5 + x_7 & x_3 + x_5 \\ x_2 + x_4 & x_1 + x_3 - x_2 + x_6 \end{bmatrix}$$

$$\begin{aligned} x_1 &= (a_{1,1} + a_{2,2}) \cdot (b_{1,1} + b_{2,2}) \\ x_2 &= (a_{2,1} + a_{2,2}) \cdot b_{1,1} \\ x_3 &= a_{1,1} \cdot (b_{1,2} - b_{2,2}) \\ x_4 &= a_{2,2} \cdot (b_{2,1} - b_{1,1}) \\ x_5 &= (a_{1,1} + a_{1,2}) \cdot b_{2,2} \\ x_6 &= (a_{2,1} - a_{1,1}) \cdot (b_{1,1} + b_{1,2}) \\ x_7 &= (a_{1,2} - a_{2,2}) \cdot (b_{2,1} + b_{2,2}) \end{aligned}$$

It's important to note that we don't use the commutativity of multiplication, and therefore it can be applied to matrices.

Suppose that we have $n = 2^k$, we can then divide up the matrix into four quadrants, each $\frac{n}{2} \times \frac{n}{2}$, and then compute $C_{i,j}$ recursively, until we finish at $n = 2$;

$$\begin{bmatrix} A_{1,1} & A_{1,2} \\ A_{2,1} & A_{2,2} \end{bmatrix} \begin{bmatrix} B_{1,1} & B_{1,2} \\ B_{2,1} & B_{2,2} \end{bmatrix} = \begin{bmatrix} C_{1,1} & C_{1,2} \\ C_{2,1} & C_{2,2} \end{bmatrix}$$

Now, if we were to consider the number of arithmetic operations $A(k)$, where $n = 2^k$.

$$\begin{aligned} &A(0) = 1 \\ &A(k) = 7A(k-1) + 18(\frac{n}{2})^2 \end{aligned}$$

Divide, and Conquer Algorithms

Strassen's was an example of a divide, and conquer algorithm, where we divide some problem into a subproblems, of size $\frac{n}{b}$. Now solve each subproblem recursively, and then combine the result at the end.

Insertion Sort

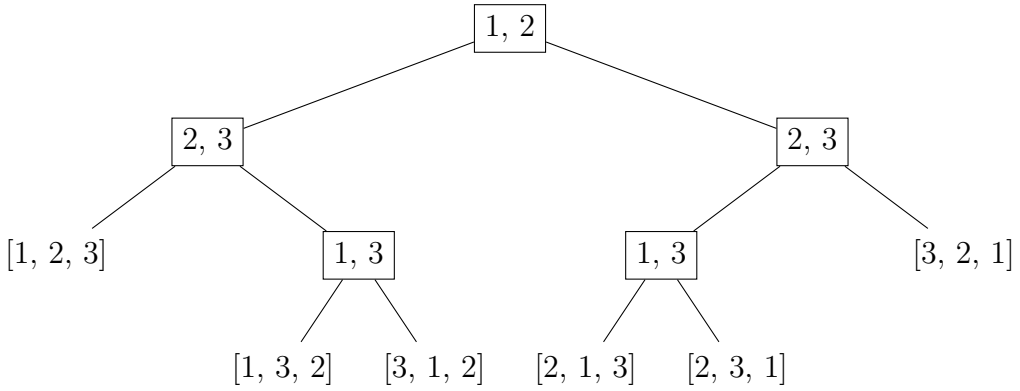
The general idea of insertion sort is to insert the item $L[i]$ into the sorted sublist $L[0..i-1]$, then it follows that the sorted sublist is now $L[0..i]$. This takes between 1, and i comparisons, getting the latter is when $L[i] < L[0]$. Therefore, it follows that the worst case for a list of size n is;

$W(n) = \sum_{i=1}^{n-1} i = \frac{n(n-1)}{2}$

This worst case is when the entire list is in reverse order, hence $W(n)$ is $\Theta(n^2)$. If we're only allowed to do local swaps, this is an optimal algorithm.

Lower Bounds for Sorting

In order to clarify some syntax, the internal nodes are comparison pairs, such that the left child of x, y, means that $L[x] \leq L[y]$, and the right child means $L[x] > L[y]$. The leaf nodes are the sorted outcomes of the trees (where the indices should map to). We are also indexing from 1, not 0.



You'll notice that each leaf node is a permutation of the list, therefore there are 3! leaf nodes. Note that the this is a 3-sort, not insertion sort. If we were to draw this for insertion sort, it would be important to list the state of the list as you make changes, to ensure we keep track of how the elements have moved. We can justify that this is optimal as any decision tree for a list of length 3 must have 3! leaves. We cannot have anything with a depth ≤ 2 as all binary trees of depth ≤ 2 have ≤ 4 leaves. Therefore it follows that the minimum bound is a depth of at least 3. In both cases, our worst-case comparisons is 3, and therefore both of these sorting algorithms are optimal for $n = 3$.

Consider a general list n. We can justify that it therefore has $n!$ permutations. Given some tree with a depth d, it follows that it has a maximum of 2^d leaves, and therefore $n! \leq 2^d$. Therefore, by arithmetic, we can justify that $d \geq \log_2(n!)$, which means that $d \geq \lceil \log_2(n!) \rceil$, since $d \in \mathbb{N}$.

21st February 2019

Average Case

When we don't know the actual distribution, we can assume that each permutation is equally likely. If we consider our previous 3-sort tree, we take the sum of the path lengths to each leaf, and then divide it by the number of permutations.

We can define the total path of a tree is the sum of the path lengths to each leaf. Suppose we have some decision tree T, for sorting a list of length n. It therefore has $n!$ leaves, and a total path length b. In the general case, we cannot change $n!$, so the average number of comparisons is $\frac{b}{n!}$, hence we want to minimise b. This is the case when we have a roughly balanced tree.

We can define a tree as being balanced when all the leaves of a tree are at either depth d, or $d - 1$. If a tree is unbalanced, we can find a balanced tree without increasing the total path length.

Merge Sort

If we divide a list roughly in two, we can then sort each half separately, by recursion, and then merge the two halves.

We merge the list by comparing the least elements of the two lists, and outputting the lowest. In the worst case, we only get one element for "free", therefore the worst case is $n - 1$ comparisons. If we make it easier for ourselves, by forcing n as a power of 2, we can get an approximation of the growth, which is good enough;

$$W(1) = 0$$
$$W(n) = n - 1 + W(\lceil \frac{n}{2} \rceil) + W(\lfloor \frac{n}{2} \rfloor)$$

Assume that $n = 2^k$

$$\begin{aligned}
W(n) &= n - 1 + 2W(\frac{n}{2}) \\
&= n - 1 + 2(\frac{n}{2} - 1 + 2W(\frac{n}{4})) \\
&= n - 1 + n - 2 + 4W(\frac{n}{4}) \\
&= 2n - 3 + 4W(\frac{n}{4}) \\
&= 3n - 7 + 8W(\frac{n}{8}) \\
&= mn - n + 1 + 2^m W(\frac{n}{2^m}) \\
&= kn - n + 1
\end{aligned}$$

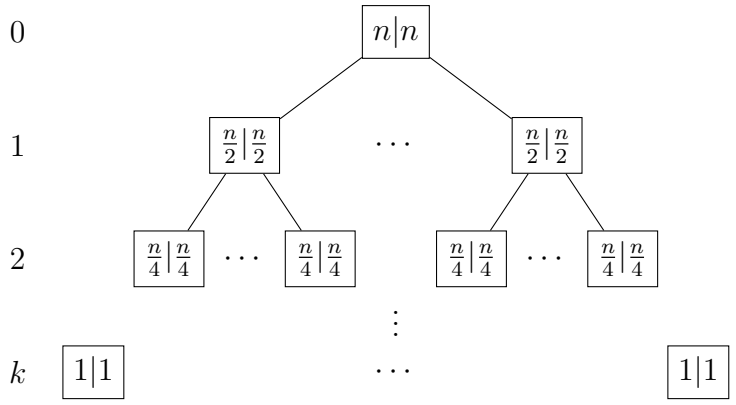
Thus $\Theta(n\log(n))$. The worst case of merge sort, and the minimum work for sorting are of the same order, therefore it is an optimum algorithm in terms of orders.

Generalised Divide, and Conquer

The general form for a DaC algorithm, is as follows; given we have work $T(n)$ for some input of size n, and we split it into a sub-problems, each of size $\frac{n}{b}$, the non-recursive work we need to do is $f(n)$, it follows that $T(n) = aT(\frac{n}{b}) + f(n)$, as well as some base cases. We will work on the following example;

$$T(1) = 1$$
$$T(n) = aT(\frac{n}{2}) + n$$

We create a recursion tree, where there is an input of size n, and each level down we, record the size $\frac{n}{2}$, and also the non-recursive work (same in this case);



Then it follows that the total work done is the sum of the work done at each level; $n + a(\frac{n}{2}) + a^2(\frac{n}{2^2}) + ... + a^{k-1}(\frac{n}{2^{k-1}}) + a^k$. However, we can easily express this as a geometric series with the ratio being $\frac{a}{2}$, and the starting term n. As we know the formula for the sum of a geometric series as follows;

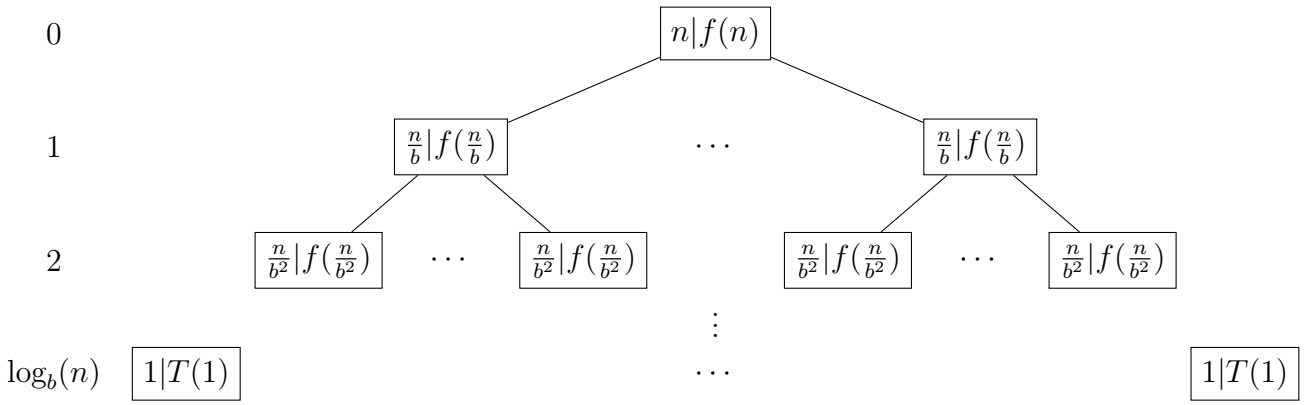
$\sum_{i=0}^{k} ar^i = \frac{a(r^{k+1}-1)}{r-1}$

We can then say that $t(n)$ is the largest term in the geometric progression $a, ar, ar^2, ..., ar^k$. Now, we can apply that to our geometric series as follows;

- if $a < 2$ the greatest term is n (non-recursive work at level 0 dominates)
  $T(n) = \Theta(n)$
- if $a = 2$ work is roughly evenly spread at all levels
  $T(n) = (k+1)n = \Theta(n\log(n))$
- if $a > 2$ the base cases (leaves of recursion tree) dominate (greatest term is $a^k = a^{\log(n)} = n^{\log(a)}$)
  $T(n) = \Theta(n^{\log(a)})$

25th February 2019

General Case of the Master Theorem



Therefore in general, it follows that the work at level 0 is $f(n)$, then at level 1 is $af(\frac{n}{b})$, 2 is $a^2f(\frac{n}{b^2})$, and so on, until level $\log_b(n)$, where the work is $\Theta(a^{\log_b(n)})$. All we need to know at the bottom level is the amount of leaf nodes, since we don't care about the constant work.

Suppose that $f(n) = n^c$, such that it is just a polynomial function, then it follows that the ratio of the work between the levels is $r = \frac{a}{b^c}$. Let us have some critical exponent $E = \log_b(a) = \frac{\log(a)}{\log(b)}$. We can then argue that $r > 1 \Leftrightarrow a > b^c \Leftrightarrow \log_b(a) > c \Leftrightarrow E > c$. Generalising from the previous section, we can say that;

- $E < c$ the work is concentrated at the top
  $T(n) = \Theta(f(n))$
- $E = c$ the work is spread through the tree
  $T(n) = \Theta(f(n)\log_b(n)) = \Theta(f(n)\log(n))$
- $E > c$ the work is concentrated at the base level
  $T(n) = \Theta(a^{\log_b(n)}) = \Theta(n^{\log_b(a)}) = \Theta(n^E)$

In order to generalise this beyond just a polynomial $f(n)$, we can state the following, as the **Master Theorem**; $T(n) = aT(\frac{n}{b}) + f(n)$ has solutions as follows; where $E = \frac{\log(a)}{\log(b)} = \log_b(a)$:

1. if $n^{E+\epsilon} = O(f(n))$ for some $\epsilon > 0$, then $T(n) = \Theta(f(n))$
2. if $f(n) = \Theta(n^E)$ then $T(n) = \Theta(f(n)\log(n))$
3. if $f(n) = O(n^{E-\epsilon})$ for some $\epsilon > 0$ then $T(n) = \Theta(n^E)$

The use of ϵ suggests that it is bounded, as we can therefore bound it in a way that suggests $n^E < n^{E+\epsilon}$.

Applying the Master Theorem

Consider the worst case number of comparisons for binary search; $W(n) = W(\frac{n}{2}) + 1$. Here, we're told that $a = 1$, $b = 2$, and $f(n) = \Theta(n^0)$. It's trivial to calculate the critical exponent here, as $E = 0$. This is the second case, as we have $f(n) = \Theta(n^E)$, therefore $W(n) = \Theta(n^0\log(n)) = \Theta(\log(n))$

Consider the worst case number of comparisons for merge sort; $W(n) = 2W(\frac{n}{2}) + (n-1)$, therefore we have $a = 2$, $b = 2$, and $f(n) = \Theta(n^1)$. We can calculate E to be 1, therefore it's once again the second case, hence $W(n) = \Theta(n\log(n))$.

Consider the number of arithmetic operations for Strassen's algorithm; $A(n) = 7A(\frac{n}{2}) + 18(\frac{n}{2})^2$. Here we have $a = 7$, $b = 2$, and $f(n) = \Theta(n^2)$. Now, we can calculate E to be $\log_2(7)$, which we can say is greater than 2, as $7 > 4$. Therefore we take the third case, as there is some positive ϵ. As such, we get the worst case as $A(n) = \Theta(n^{\log(7)})$. Note that any improvement to $f(n)$ will not help with the order of the complexity.

Quicksort

In quicksort, we pick some pivot element (take the first element for simplicity). We then split it into two sublists, where the items on the left side of the pivot are less than it, and the ones on the right are greater than it. The first split clearly takes $n-1$ comparisons. The pseudocode for the split is as follows;

```
split(left, right):
  d = L[left] # pivot
  i = left + 1
  j = right
  while i <= j:
    if L[i] <= d:
      i = i + 1
    else:
      swap(i, j)
      j = j - 1
  swap(left, j) # swaps the pivot into the middle of the list
  return j
```

In the worst case it is when the list is already sorted, and it essentially becomes the worst case of insertion sort. However, we quicksort is still good in practice, since we are unlikely to find the split at either end. Given we split on some position s, from a list indexed 1 to n inclusive, then it follows that we are doing the sort on two sublists, of sizes $s-1$, and $n-s$. Therefore it takes $A(s-1)+A(n-s)$ comparisons, on the sublists;

$A(n) = n - 1 + \frac{1}{n}\sum_{s=1}^{n}(A(s-1)+A(n-s))$

This can be simplified to the to the following recurrence relation;

$$A(1) = 0$$

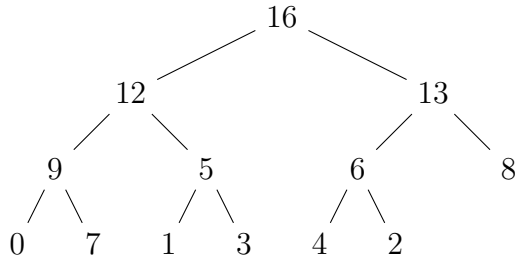
$$A(n) = n - 1 + \frac{2}{n}\sum_{i=2}^{n-1} A(i)$$

This can be solved such that the average case becomes $\Theta(n\log(n))$. No idea how, but just trust it, I guess.

While it seems that merge sort seems to be much better than quicksort, the latter can improve the chances of a good split, whereas merge sort is required to split down the middle, and the latter also requires less memory since it can all be done in place.

Heapsort

A **heap structure** is a left-complete binary tree. For a tree with depth d, to be left-complete; all nodes are present at depths 0 to $d-1$, and at d, no node is missing to the left of a node which is present. The rightmost tree at d is the last node. A tree T is a minimising partial order tree if the key at any node is $\leq$ the keys at each child node. A min heap is a heap structure that combines this property. A max heap is similar, but with $\geq$ the child nodes. The following is an example of a max heap;



```
procedure sort():
  # build a max heap H out of an array E of   elements
  for i = n to 1:
    max = getMax(H)
    deleteMax(H)
    E[i] = max
procedure getMax(H):
  # read root node of H
procedure deleteMax(H):
  # copy element at last node into root node
  # remove last node
  fixMaxHeap(H)
procedure fixMaxHeap(H):
  # O(log n)
  if not leaf(H):
    largerSubHeap = left or right subheap with larger root
    if root(H).key < root(largerSubHeap).key:
      swap root(H) and root(largerSubHeap)
      fixMaxHeap(largerSubHeap)
```

Suppose that $n = 2^k - 1$, therefore the heap structure is a complete binary tree with a depth of $k - 1$. Let $W(n)$ be the worst-case number of comparisons for `buildMaxHeap`; $W(n) = 2W(\frac{n-1}{2}) + 2\log(n)$. By application of the master theorem, we have $a = 2$, $b = 2$, and $f(n) = \Theta(\log(n))$. Then it follows that $E = 1$, therefore it's the third case, and $T(n) = \Theta(n)$. Therefore, building the heap is a linear time operation. As such, we the overall comparison count is; $O(n\log(n))$.

While the tree representation of trees is quite helpful to visualise the concept, we can implement a heap using arrays. Starting at 1 allows for easier arithmetic. Given some node at index i, we store the left, and right children at $2i$, and $2i + 1$ respectively. This then allows us to get the parent node of any node at index i as $\lfloor \frac{i}{2} \rfloor$. Without using pointers, we can reduce the overhead, and it allows us to carry out heapsort in place just like quicksort. We can store the heap previously drawn in an array as follows, using the rules previously mentioned;

index	1	2	3	4	5	6	7	8	9	10	11	12	13
value	16	12	13	9	5	6	8	0	7	1	3	4	2

```
procedure heapsort(E, n):
  buildMaxHeap(1, n)
  heapsize = n
  while heapsize > 1:
    swap(1, heapsize) # put the root into the sorted list
    heapsize = heapsize - 1
    fixMaxHeap(1, heapsize)
procedure buildMaxHeap(root, heapsize):
  # not actually sure this is correct
  left = 2 * root
  right = 2 * root + 1
  if left <= heapsize:
    buildMaxHeap(left, heapsize)
    buildMaxHeap(right, heapsize)
    fixMaxHeap(root)
procedure fixMaxHeap(root, heapsize):
  left = 2 * root
  right = 2 * root + 1
  if left <= heapsize:
```

```
    # the root isn't a leaf
    if left = heapsize:
      # no right subheap
      largerSubHeap = left
    else if E[left].key > E[right].key:
      largerSubHeap = left
    else:
      largerSubHeap = right
    if E[root].key < E[largerSubHeap].key:
      swap(root, largerSubHeap)
      fixMaxHeap(largerSubHeap, heapsize)
```

28th February 2019

Priority Queues

Now that we have some knowledge on what a heap is, we can implement a priority queue as a binary heap. For Prim's algorithm, we use a min heap. The operations are the same - but we will also include how they are implemented with the heap.

PQ operation	complexity	heap action
`Q = PQCreate()`		create an empty array `E` of a suitable size, and `heapsize` $= 0$
`isEmpty(Q)`	$O(1)$	check if `heapsize` $= 0$
`getMin(Q)`	$O(1)$	return `E[1]`
`deleteMin(Q)`	$O(\log(n))$	`E[1] = E[heapsize]; heapsize = heapsize - 1;` `fixMinHeap(1, heapsize)`
`insert(Q, x)`	$O(\log(n))$	`heapsize = heapsize + 1; E[heapsize] = x;` `percolateup(heapsize)`

```
procedure percolateup(c):
  if c > 1:
    parent = floor(c / 2)
    if E[c].key < E[parent].key
    swap(c, parent)
    percolateup(parent)
```

The one function we haven't considered yet is `decreaseKey(Q, x, newKey)`. If we know the location, let it be `c`, of `x` in the heap, then we can change the key to `newKey` and run `percolateup(c)`. A solution would be to give each element an ID. We can then take a supplementary array `xref` to store the location of an ID. Therefore, we can use `xref[id] = k` means that the element is at location `k` in the heap. However, we will need to modify `percolateup` to update `xref`.

Word Break Problem

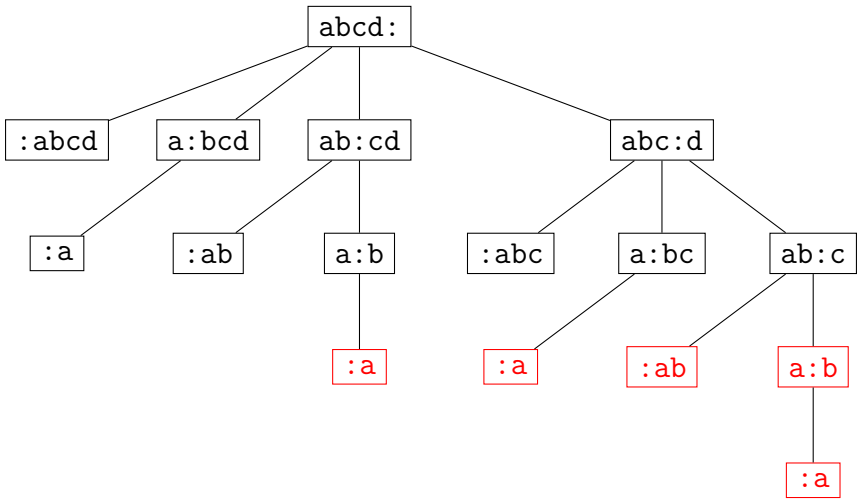
While we've already seen some examples of dynamic programming, we should consider examples to illustrate the types of solutions; **top-down**, and **bottom-up**.

Given some string of characters `s`, can we split it into words occurring in a dictionary? For example; the "windown" can be split into "win down", or "wind own" (hence there isn't necessarily a unique solution), whereas the string "trcarlenz" cannot be split. If we were to consider all possible splits, that would take exponential time. A **top-down** recursive solution would be as follows;

```
# s[i:j] takes strings from index i to j - 1
# s[:j]  takes strings from index 0 to j - 1
# s[i:]  takes strings from index i to len(s) - 1
```

```
procedure wb1(s):
  if len(s) == 0:
    return true
  else:
    for i = 0 to len(s) - 1:
      if indict(s[i:]):
        if wb1(s[:i]):
          return true
  return false
```

By solving a recurrence relation, we can see that the complexity of this is $W(2^n)$. We can consider the operation of this as a tree (formatted as `RECURSIVE CALL : DICT LOOKUP`). You'll notice that the nodes in red are being recomputed, even though they have already been computed before;



In order to remove the "red nodes", we can use a memoised recursive solution. We're trading off space for time, since we're storing values we've already computed.

```
memo = []
procedure wb2(s):
  if len(s) == 0:
    return true
  else:
    for i = 0 to len(s) - 1:
      if indict(s[i:]):
        if memo[s:[:i]] undefined:
          memo[s:[:i]] = wb2(s[:i])
        if memo[s:[:i]]:
          return memo
  return false
```

We then end up with a recurrence relation here;

$$\begin{aligned} W_2(0) &= 0 \\ W_2(n) &= n + W_2(n-1)(n \geq 1) \\ &= \frac{n(n+1)}{2} \end{aligned}$$

Another method is to take the non-recursive bottom-up approach, similar to what we did for the Bellman-Held-Karp algorithm, by solving increasing subproblems. Thus we're solving `s[:0], s[:1], ..., s[:n] = s`. Note that we're storing the result for `s[:i]` in `wb[i]`.

```
procedure wb3(s):
  n = len(s)
```

```
  wb[0] = true
  if n > 0:
    for i = 1 to n:
      wb[i] = false
      for j = 0 to i - 1:
        if wb[j] and indict(s[j:i]):
          wb[i] = true
          break
  return wb[n]
```

Comparing `wb2`, and `wb3`, they each have their benefits. The former is easier to develop, and might be faster if we don't need to work out the subproblems. On the other hand, the latter has less overhead from the recursive calls.

Dynamic programming aims to reduce a problem with an exponentially large solution, and find a path to a solution in polynomial time. We use the term programming to plan an order of computation, and the subproblems we consider depend on our previous results (hence it is dynamic).

4th March 2019

Tractability

We want to identify which problems are feasible, therefore can be computed in a reasonable amount of time. When we do this, we should be focusing on the worst-case, such that $W(n)$ doesn't grow "too large". We've proven that we can solve sorting in polynomial time with multiple different algorithms, but consider more complex graph problems.

Given some graph G, we want to know whether there exists an Euler path. The input size of G, depends on how we represent it; either as an adjacency matrix of size $O(n^2)$, or as a linked list with size $O(n+m)$. From before, we know that G has an EP $\Leftrightarrow$ it has 0, or 2, nodes of odd degree. This is a trivially tractable problem, as counting the number of odd degree nodes is extremely easy.

In contrast, checking whether a Hamiltonian path exists within some graph G is not tractable. This is because it requires $O(n^2 2^n)$ time, which is too slow.

In this course, we focus mostly on decision problems, such that a decision problem D is decided by some algorithm A, which returns 'yes', or 'no', for any input x. A must terminate. A decision problem D is in p-time if there exists some algorithm that solves it in p-time, such that on any input size, A will take $\leq p(n)$ for some polynomial p. We will use D to range over decision problems, as P exists as the class of p-time problems. Cook-Karp Thesis states that a problem is tractable $\Leftrightarrow W(n) \leq p(n)$ (can be computed within polynomially many steps in a worst case). While the use of p suggests that we can even have a polynomial of order 1000, realistically most algorithms would run within order 10.

We discuss $p(n)$ in terms of some input size n, and an arbitrary computation step. In our list sorting examples, we just take n to be the length of the list, and only counted comparisons (thus ignoring other computation steps, such as recursive overhead, memory, swaps etc.). It turns out that all reasonable models give the same results.

The polynomial invariance thesis states that if some problem can be solved in $p(n)$ in some reasonable model, changing the model to another reasonable model would mean that the problem can still be solved in $q(n)$, where q is another polynomial.

We will define a decision problem as being in the complexity class P if it can be decided within polynomial time$p(|x|)$ in some reasonable model of computation. Note that sorting a list is not in this class P, as it's not a decision problem.

An unreasonable model would be something along the lines of being able to carry out more than polynomially many operations in parallel, since we would then be able to solve an exponential problem in p-time (superpolynomial parallelism), see quantum computation. The use of unary numbers is

also unreasonable, as it gives us exponentially larger input size, and therefore some exponential time algorithm can appear as p-time, since we've artificially scaled up the input size.

It's important to note that the arithmetical operations are all in p-time.

I/O Size

Suppose that we have f, a p-time function. The output size $|f(x)|$ is polynomially bounded in the input size $|x|$, such that $|f(x)| \leq p(|x|)$. This is argued by the fact that any program which computes f, is limited by having p-time to build the output.

Function Composition

Suppose that we have f, g both p-time computable functions. Then the composition $g \circ f$ is also p-time computable. Suppose that $f(x)$ is computed by A, within $p(n)$, where $n = |x|$, and $g(y)$ by B, within $q(m)$, and $m = |y|$.

In order to compute the composed function, we'd need to first run A on x. By definition it must then run in $\leq p(n)$ steps. And therefore the maximum output it can generate is also polynomial. hence $|f(x)|$ must be polynomially bounded in n, such that we can say $|f(x)| \leq p'(n)$ for some polynomial $p'(n)$. Then it follows that B will run within $q(p'(n))$ steps. Hence, the total running time of the function, A followed by B, is $p(n) + q(p'(n))$. This result is polynomial in n.

Guessing a Certificate

Going back to the Hamiltonian Path problem, and a graph G. Let us guess some path π. Checking that π is a HP of G is trivial; we need to check that the items of π are a permutation of nodes(G), and then check that successive nodes in π are adjacent in G. This can easily be done in p-time.

The problem with this is that, if we guess a path correctly, then the problem is trivial to solve in p-time, however, if we guess it incorrectly, such that π is not a HP of G, then we aren't closer to the solution (maybe we're able to exclude π from the search space now - if it's the last permutation to check, then it's useful). We still don't know whether it has a path or not.

Let us define some problem VHP (verify HP), taking 2 inputs; G, and some path π. Note that this verification problem runs in P. Trivially we can say that $\text{HamPath}(G) \Leftrightarrow \exists\pi[\text{VerHamPath}(G, \pi)]$.

We can now define a decision problem $D(x)$ as being in NP (non-deterministic polynomial time) if there is a problem $E(x, y)$ in P, and a polynomial $p(n)$ such that $D(x) \Leftrightarrow \exists y[E(x, y)]$, and if $E(x, y)$ then $|y| \leq p(|x|)$ (this means that the guess / certificate is polynomially bounded in the input size). The second condition is required, as it would take too long to even guess y if we didn't have that restriction. In general, we can say that P is the class of decision problems, which can be efficiency **solved**, and NP is the class of decision problems that can be efficiency **verified**.

7th March 2019

Satisfiability

A formula ϕ is in CNF (from **CO140**) if it is of the form;

$\bigwedge_i (\bigvee_j a_{ij})$

Where each term a_ij is either a variable x or it's negation $\neg x$. The terms a_{ij} are called literals, and the terms $\bigvee_j a_{ij}$ are called clauses. The SAT problem (throwback to Haskell exam), given a formula ϕ in CNF, is it satisfiable? Is there some assignment v to the variables of ϕ, which makes ϕ true?

It seems that SAT is not something we can solve in p-time, since we'd have to try all possible assignments. Let $n = |\phi|$, where we get the number of symbols in ϕ, and $|v|$ be m, the size of the domain of v. Can we test that SAT belongs to the class NP? Let us guess some truth assignment v, and we can

verify in p-time that it satisfies ϕ. Let there be $\text{VSAT}(\phi, v) \Leftrightarrow \phi$ is in CNF, and v satisfies ϕ. Then it follows that $\text{SAT}(\phi) \Leftrightarrow \exists v[\text{VSAT}(\phi, v)]$. If $\text{VSAT}(\phi, v)$, then $|v| \leq |\phi|$.

P vs. NP

Suppose we have some decision problem D, which we know is P. To verify that $D(x)$ holds, we don't need to guess some y, we can directly decide it. However, let's define $E(x, y) \Leftrightarrow D(x) \wedge y = \epsilon$, where the epsilon is just a dummy guess. Then clearly, we can say $D(x) \Leftrightarrow \exists y[E(x, y)]$, and $|y| \leq p|x|$. Hence we've proven that $\text{P} \subseteq \text{NP}$.

Whether we can say $\text{P} = \text{NP}$ is still unknown. While the majority of researchers believe $\text{P} \neq \text{NP}$, it remains as one of the most important open problems in mathematics, let alone in computer science.

Problem Reduction (P)

Since we know that $\text{P} \subseteq \text{NP}$, showing a problem belongs to NP doesn't necessarily mean it's not tractable (in P). We want to identify high complexity (hard) problems in NP. Consider two.

Suppose that we have two decision problems D, and D'. We can say that D reduces to D' (such that $D \leq D'$) if there exists some p-time (possibly many-to-one) function f, such that $D(x) \Leftrightarrow D'(f(x))$.

Suppose there is some algorithm A', which decides D' in time $p'(n)$. Then it follows if $D \leq D'$ via some reduction function f running in $p(n)$, we have some algorithm A which decides D. We define A on the input x, such that we first compute $f(x)$, and then run $A'(f(x))$, and return that answer. This also runs in p-time, under the same argument we made when composing p-time functions.

Suppose $D \leq D'$ and $D' \in \text{P}$, then $D \in \text{P}$.

Problem Reduction (NP)

Suppose $D \leq D'$ and $D' \in \text{P}$, then $D \in \text{NP}$.

Given the first assumption, we can assume that there is some $E'(x, y) \in \text{P}$, and a $p'(n)$ (also p-time) such that $D'(x) \Leftrightarrow \exists y[E'(x, y)]$. And if $E'(x, y)$ then it follows that $|y| \leq p'(|x|)$, also by definition of being reducible, we have $D(x) \Leftrightarrow D'(f(x))$. By combining this, we can say the following;

$D(x) \leftrightarrow \exists y[E'(f(x), y)]$, and we can trivially define a polynomial verification function $E(x, y) \Leftrightarrow E'(f(x), y)$. As such, we have $D(x) \Leftrightarrow \exists y[E(x, y)]$. Checking that E is p-balanced is also done under the same argument. Suppose we have $E(x, y)$, then $E'(f(x), y)$ so that $|y| \leq p'(|f(x)|)$ However, we know that $|f(x) \leq q(n)|$, hence it follows that $|y| \leq p'(q(|x|))$, therefore $D \in \text{NP}$.

Note that the reduction order is reflexive, and transitive such that $D \leq D$, and if $D \leq D' \leq D''$ then $D \leq D''$. For the former, we can prove it by letting the reduction function be $f(x) = x$, and the latter is (probably) done with polynomial function composition, as that keeps it closed within the set of polynomial functions. If both $D \leq D'$, and $D' \leq D$, then we can write $D \sim D'$, which states that they are as hard as each other.

NP-completeness

We identify a decision problem D as being NP-hard if for all problems $D' \in \text{NP}$, we have $D' \leq D$, such that we can reduce D' to D. Such an NP-hard problems do not necessarily belong to NP, and that they could be harder. We can state a problem as being NP-complete, if it is an NP problem, and it is NP-hard (such that $D \in \text{NP}$).

We take SAT as an NP-complete problem (proved). Instead of proving it directly, we can reduce it as follows. In order to see that D is NPC, we need to show that $D \in \text{NP}$, and that $D' \leq D$ for some NPC complete D'. The latter criteria establishes that D is NP-hard, given that D' is also NP-hard.

For example, let us show that the Hamiltonian Path problem is NP-complete. We've already verified it being $\in$ NP, as we've done the proofs for the guess, and p-time verification. It's possible to show that $\text{SAT} \leq \text{HP}$ (long proof), therefore HP is NP-complete.

Intractability via NP-completeness

Suppose we have P $\neq$ NP, if some problem D is NP-hard, then $D \notin$ P. The proof is as follows;

Assume that P $\neq$ NP, and that D is NP-hard. By contradiction, let us assume $D \in P$. Take another problem $D' \in$ NP, but as we know D is NP-hard, we have $D' \leq D$, therefore it follows that $D' \in$ P. As such, we've shown NP $\subseteq$ P, but given that P $\subseteq$ NP, it follows that P = NP. This contradicts our assumption.

This means that if we show a problem is NPC, we know that it is intractable (as long as P $\neq$ NP).

14th March 2019

Travelling Salesman Problem

While the problem is fairly well understood, we cannot deal with it since it's an optimisation problem, and not a decision problem. A common method for doing this is to give it some upper bound, such that the problem then becomes "given a weighted graph (G, W), and a bound B, is there a tour of G with a total weight $\leq B$?"

Now, we can try, and show that TSPD is NP-complete, by fulfilling the two criteria;

1. TSPD $\in$ NP

   let us guess some path π, we can trivially check that π is a Hamiltonian circuit of G, and that $W(\pi) \leq B$. We know that π is obviously bounded, hence $|\pi| \leq |G|$.

   We can formally define VTSPD$((G, W), B, \pi) \Leftrightarrow$ VHP$(G, \pi) \wedge W(\pi) \leq B$ - note that in this case, we are using VHP(G, π) to mean that π is a Hamiltonian path of G.

2. $D' \leq$ TSPD

   see below

HamPath $\leq$ TSPD

We need to then define some p-time function f which is able to transform a graph G into some weighted graph (G', W), with some bound B. To construct the weighted graph (G', W), we do the following. First let nodes(G') = nodes(G). Given a pair of distinct nodes x, y in G, if $(x, y) \in$ arcs(G), then $W(x, y) = 1$. Otherwise $W(x, y) = 2$. Essentially, we're adding in the arcs with higher weight (since we want a connected graph for TSP). Then we set the upper bound $B = n + 1$, where n is the number of nodes in G. This reduction is trivial to see as a polynomial time function, $f(G) = ((G', W), B)$.

However, to show that f is a valid reduction, we need to verify HP$(G) \Leftrightarrow$ TSPD$((G', W), B)$. Suppose that G has a HP π, with endpoints x, and y. If we follow that path in G', it has a weight of $n - 1$, as we have n nodes, thus $n - 1$ arcs (we can't really repeat an arc, as we'd end up going back to a node). We get a TS tour by adding in the arc (x, y), with $W(x, y) \leq 2$, by our definition of f. Therefore, the tour weight $\leq n + 1 = B$.

To prove the other direction of the implication, suppose that we have some weighted graph (G', W), which has a tour of weight $\leq B = n = 1$. This has n arcs, since we're not revisiting nodes. At most, only one arc can have a weight of 2 (any more would lead to the path being over-budget). Now, if we were to remove that that arc, we'd have a Hamiltonian path in G. Since we've proven both sides of the implication, we've then proven that HP $\leq$ TSPD, thus TSPD is an NP-hard problem, that is also NP, hence it is NP-complete.

In order to show that it is intractable (assuming P $\neq$ NP), we prove it by contradiction. Let us suppose that it can be solved by a p-time algorithm, such that the optimal time O can be computed in p-time. Therefore, it follows that TSPD is p-time, since we just need to check whether the path is valid (it would be, with a correct algorithm), and then check that $O \leq B$. However, this wouldn't be possible, since we have just proven that TSPD is NP-complete, and that P $\neq$ NP.

Metric TSP

The MTSP restricts to graphs (G, W) that satisfy the triangle inequality. Such that $\forall x, y, z \in \text{nodes}(G)[W(x, z) \leq W(x, y) + W(y, z)]$. This is a fairly natural condition, as if it didn't hold, we'd travel via y. Once again, we can make a decision version of it; MTSPD. The method for checking the first criteria is the exact same as for TSPD, where we have some verification function. We cannot easily do this reduction with the TSP, since we cannot do the implication the other way (since we're reducing the weights of the graphs, having it remain in budget in from TSP to MTSP is trivial, but proving it the other way can be extremely challenging).

Instead, we look to reduce the HP problem to MTSPD. We'll apply the same rules for the weighting function from the HP problem to the TSPD problem for MTSPD. We know that $1 \leq W(x, y) \leq 2$ for all $x \neq y$. Therefore, it follows that $W(x, z) \leq W(x, y) + W(y, z)$; since it's an equality, let us consider the worst case on either side, the greatest possible value of the LHS, and the least possible value on the RHS. Because it is bounded, we know the greatest value of the LHS is 2. For the RHS, we know the least value is 2 $(1 + 1)$, therefore in any case $2 \leq 2$. By applying the same justifications as we did for HP to TSPD, we can conclude that HP $\leq$ MTSPD.

Vehicle Routing Problem

Suppose that we have a depot, some vehicles, and various deliveries (with destination addresses). We can model this as a complete graph, where the nodes are the depot, and the customer addresses. This graph must satisfy the triangle inequality, and $W(x, y)$ once again represents the cost of the shortest path from x, to y. We have additional constraints in that the vehicles have a capacity, and that the sum of the sizes of the packages transported on each trip cannot exceed this capacity. We can also assume that each vehicle does at least one trip.

The problem (VRPCD) is now as follows, where we are given;

- a complete weighted graph (G, W) which satisfies the triangle inequality
- a distinguished `start` node (the depot)
- k vehicles, each with capacity `C`
- a set of packages of size $s_1, ..., s_n$, to be delivered to nodes $x_1, ..., x_n$ respectively.
- a budget B

Can we deliver the items within the total cost? We want to show that this is NP-complete, and by the method; we need to do the following;

1. we can show that VRPCD $\in$ NP, by finding a simple verification algorithm that runs in p-time with a given input
2. find some D', where $D' \leq$ VRPCD

   see below

MTSPD $\leq$ VRPCD

Due to the use of the triangle inequality, we can instead use MTSPD, and show that MTPSD $\leq$ VRPCD. Given a graph (G, W), with n nodes satisfying the triangle inequality, and a budget B from the MTSPD implementation, we keep (G, W), and B the same. We mark one node to be the `start` (depot). Now we can assign one package, with size 1, to each of the non-depot nodes, and create a single vehicle with a capacity of $n - 1$. All of this can be easily done in p-time.

Now we just have to verify that MTSPD$((G, W), B) \Leftrightarrow$ VRPCD$(f((G, W), B))$.

Given that MTSPD$((G, W), B)$, this tells us that we can start at some `start` node, and visit each node exactly once, and return, within the cost B. As the graph is unaltered, we an do the same in the

VRPCD analogue. We also don't exceed the vehicle capacity since we've set the value, such that the capacity matches exactly.

To prove the implication the other way, we assume VRPCD($f((G, W), B)$). Then it follows that we have a route that starts at the depot, and delivers packages to each node (hence it must visit each node), and return to the depot within the cost B. This is a TS tour, and we can justify that we won't visit a node more than once since the graph satisfies the triangle inequality. Hence MTSPD($(G, W), B$).