Imperial College
London

NOTES

IMPERIAL COLLEGE LONDON

DEPARTMENT OF COMPUTING

Discrete Maths

Author:
Your Name (CID: your college-id number)

Date: April 15, 2017

1 Graphs and Graphs Algorithm

1.1 Graphs

Definition 1.1.1

A **graph** consists of points and lines between points. Points are called vertices and lines are called edges. A simple graph has no parallel arcs (two arcs with same endpoints) and loops (arc with both endpoints the same).

Definition 1.1.2

The **degree of a node** is the number of arcs incident on it, where loops are counted twice. A node is said to be odd (even) if its degree is odd (even).

Theorem 1.1.3

In any graph, the total of the degrees of all the nodes is twice the number of arcs.

Definition 1.1.4

Let G_1, G_2 be graphs. G_1 is a **subgraph** of G_2 if $nodes(G_1) \subset nodes(G_2)$ and if $arcs(G_1) \subset arcs(G_2)$.

1.2 Graph Isomorphism and Planar Graphs

Definition 1.2.1

Let G_1, G_2 be graphs. An **isomorphism** from G_1 to G_2 is a bijection $f : nodes(G_1) \to nodes(G_2)$ together with a bijection $g : arcs(G_1) \to arcs(G_2)$ such that for any arc $a \in arcs(G_1)$, if the endpoints of a are $n1, n2 \in nodes(G_1)$, then the endpoints of $g(a)$ are $f(n1)$ and $f(n2)$.

To test whether two graphs are isomorphic:

1) do they have the same number of nodes?

2) do they have the same number of arcs?

3) do they have the same number of loops?

4) do their nodes have the same degrees (possibly reordered)?

If they fail tests of this kind then they can?t be isomorphic.

Definition 1.2.2

Let G be a graph. An **automorphism** on G is an isomorphism from G to itself.

Definition 1.2.3

Two graphs are **homeomorphic** if they can be obtained from the same graph by a series of operations replacing an arc $x-y$ by two arcs $x-z-y$.

Theorem 1.2.4

A graph is **planar** iff it does not contain a sub-graph homeomorphic to K_5 or $K_{3,3}$

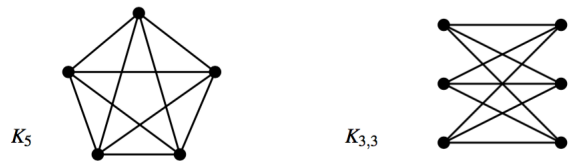

Figure 1.7: Two non-planar graphs

Theorem 1.2.5

Let G be a connected planar graph. Let G have N nodes, A arcs and F faces. Then $F = A - N + 2$.

Definition 1.2.6

A graph G is **k-colourable** if $nodes(G)$ can be coloured using no more than k colours, in such a way that no two adjacent nodes have the same colour.

Definition 1.2.7

A graph G is **bipartite** if $nodes(G)$ can be partitioned into two sets X and Y in such a way that no arc of G joins any two members of X, and no arc joins any two members of Y .

Theorem 1.2.8

A graph is bipartite iff it is 2-colourable.

Theorem 1.2.9

Every map (equivalently, every simple planar graph) is 4-colourable.

1.3 Euler Paths and Hamiltonian Circuits

Definition 1.3.1

A graph is **connected** if for any two different nodes n, n' there is a path from n to n' (n and n' need not be joined directly).

Definition 1.3.2

An **Euler path** is a path which uses each arc exactly once.

Theorem 1.3.3

A connected graph has an **Euler path** iff the number of odd nodes is either 0 or 2. A connected graph has an Euler circuit iff every node has even degree.

Proof

Suppose that a graph G has an Euler path P starting at node n_0 and ending at n_k. If P passes through a node n, it must enter n as many times as it leaves it. Hence the degree of n must be even. If n has odd degree, then P cannot use all arcs incident on n (there must be at least one left over). So every node of G must have even degree except for n_0 and n_k.

By similar reasoning, if n_0 and n_k are different then they must have odd degree, and if they are the same (so that P is in fact a cycle) then $n_0 = n_k$ must have even degree.

Definition 1.3.4

An **Euler Circuit** is 1) a cycle which uses all the arcs of the graph 2) uses each arc exactly once.

Definition 1.3.5

A **Hamiltonian path** is a path which visits every node in a graph exactly once.

Definition 1.3.6

A **Hamiltonian circuit** is a cycle which visits every node exactly once.

1.4 Trees

Definition 1.4.1

A **rooted tree** consists of points and lines between the points. The points are called the nodes and the lines are edges of the tree. A tree has the following properties:
1) There is a special node called the root
2) Every node x apart from the root is connected to a node y
3) Every node x apart from the root has exactly one parent
4) Every node has a path to the root

Theorem 1.4.2

Let T be a tree with n nodes. Then T has $n-1$ arcs

Definition 1.4.3

Let G be a graph. An nonrooted tree T is said to be a **spanning tree** for G if T is a subgraph of G and $nodes(T) = nodes(G)$.

Theorem 1.4.4

Let G be a connected graph. Then G has a spanning tree.

Proof

If G is a tree then the theorem is true. Otherwise, G has a cycle $v_0, v_1, v_2, ..., v_n, v_0$. Consider the new graph G_1 with the same vertices and the same set of edges but without the edge $e = (v_0, v_n)$.

We want to show that G_1 is a connected graph. Let x and y be two arbitrary vertices in G and there is a path in G, $x, q_1, q_2, ...y$. We want to show that there is a path from x to y in G_1. If this path does not use the edge $e = (v_0, v_n)$, then there will still be a path from x to y in G_1. If this path $x, q_1, q_2, ...y$ uses the edge $e = (v_0, v_n)$, We can transform the path $x, q_1, q_2, ...v_0, v_n, ...y$, to $x, q_1, q_2, ...v_0, v_1, v_2, ...v_n, ...y$. We can apply this process repeatedly to all occurrences of e in G. ThusG_1 is connected graph.

We can repeat the process to remove all cycles in G to produce a spanning tree.

1.5 Directed Graphs

Definition 1.5.1

A **directed graph** consists of points and directed lines connecting the points. Points are usually called vertices and directed lines are called edges.

Definition 1.5.2

The **indegree** of a node x is the number of arcs entering x. The **outdegree** of a node x is the number of arcs leaving x.

1.6 Graph Traversal

Definition 1.6.1

In **depth-first search** (DFS) we start from a particular node start and we continue outwards along a path from start until we reach a node which has no adjacent unvisited nodes. We then backtrack to the previous node and try a different path.

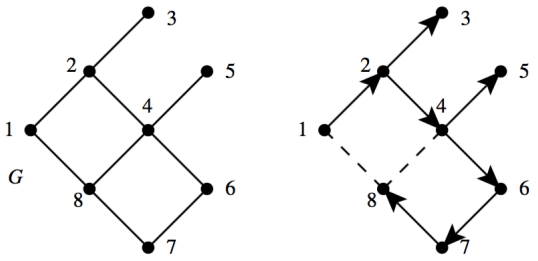

Figure 1.16: Depth-first Search

Algorithm 1.6.2

Algorithm Depth-first Search (DFS):
procedure dfs(x):
  visited[x] = true
  print x
  for y in adj[x]:
    if not visited[y]:
      parent[y] = x
      dfs(y)
      # backtrack to x

Definition 1.6.3

In **breadth-first search** (BFS) we start from a particular node start and we fan out from there to all adjacent nodes, from which we then run breadth-first searches.

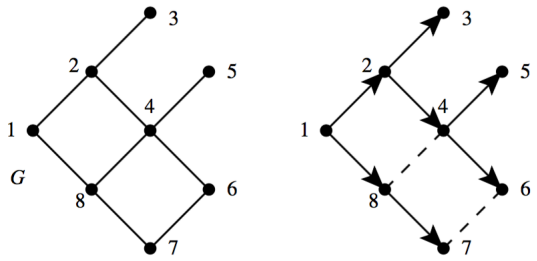

Figure 1.17: Breadth-first Search

Algorithm 1.6.4

Algorithm Breadth-first Search (BFS):
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)

Theorem 1.6.5

A graph is connected iff the list produced by DFS or BFS is the same as the complete list of nodes.

Theorem 1.6.6

Suppose a connected graph has n nodes. If it has $\geq n$ arcs then it contains a cycle.

Theorem 1.6.7

Let G be a connected graph, and let T be a spanning tree of G obtained by DFS starting at node start. If a is any arc of G (not necessarily in T), with end points x and y, then either x is an ancestor of y in T or y is an ancestor of x in T.

Proof

Suppose that we visit x before visiting y. Then we must visit y from x directly (x is the parent of y) unless we have already visited y via calling DFS on some other node z adjacent to x. In either case x is an ancestor of y. Similarly, if we visit y before x then y is an ancestor of x.

Theorem 1.6.8

Suppose that we are using DFS to traverse a connected graph. If when at node x we encounter a node y which we have already visited (except by backtracking), this tells us that the node can be approached by two different routes from the start node. Hence there is a cycle in the graph.

Algorithm 1.6.9

Algorithm Test for Cycles Using DFS:

```
procedure cycleDfs(x):
   visited[x] = true
   # print x
   for y in adj[x]:
      if visited[y] and (parent[x] undefined or y ≠ parent[x]):
         # cycle found involving x and y
         return (x,y)
      if not visited[y]:
         parent[y] = x
         cycleDfs(y)
         # backtrack to x
return "acyclic"
```

Algorithm 1.6.10

```
Algorithm Shortest Path (Unweighted Graph):
visited[x] = true
distance[x] = 0
enqueue(x, Q)
while not isempty(Q):
   y = front(Q)
   for z in adj[y]:
      if not visited[z]:
         visited[z] = true
         parent[z] = y
         distance[z] = distance[y] + 1
         enqueue(z, Q)
   dequeue(Q)
```

1.7 Weighted Graphs

Definition 1.7.1

A **weighted graph** is a simple graph G together with a weight function $W : arcs(G) \to \mathbb{R}^+$

Definition 1.7.2

Let G be a weighted graph. The weight of a spanning tree T for G is the sum of the weights of the arcs of T. T is a **minimum spanning tree** (MST) for G if T is a spanning tree for G and no other spanning tree for G has smaller weight.

Algorithm 1.7.3

```
Algorithm Prim's MST Algorithm:
Choose any node start as the root
tree[start] = true
for x in adj[start]:
   # add x to fringe
   fringe[x] = true
   parent[x] = start
   weight[x] = W[start, x]
while fringe nonempty:
   Select a fringe node f such that weight[f] is minimum
   fringe[f] = false
   tree[f] = true
   for y in adj[f]:
      if not tree[y]:
         if fringe[y]:
            # update candidate arc
            if W[f, y] < weight[y]:
               weight[y] = W[f, y]
               parent[y] = f
         else:
            # y is unseen
            fringe[y] = true
            weight[y] = W[f, y]
            parent[y] = f
```

Proof

Let G be a connected weighted graph. We want to show that Prim?s algorithm constructs an MST for G.

Let G have n nodes. Each stage of the algorithm adds an arc to the subgraph constructed so far. Let the subgraphs constructed at each stage be $T_0,...,T_k,....$ We start with T_0 having just the node start. Let T_{k+1} be got from T_k by adding arc a_{k+1}. We shall show by induction on k that each T_k is a subgraph of an MST T' of G.

Base case $k = 0$. T_0 has one node and no arcs. Clearly $T_0 \subset T'$ for any MST T' of G.

Assume that $T_k \subset T'$, some MST T' of G. Let the new arc a_{k+1} join node x of T_k to a new fringe node y not in nodes(T_k).

If $a_{k+1} \in arcs(T')$ then $T_{k+1} \subset T'$ as required.

If $a_{k+1} \notin arcs(T')$. Since T' is a spanning tree, there must be a path P from x to y. Then there must be an arc a in P which joins a node x' of T_k to a fringe node y' not in nodes(T_k).

Since the algorithm chooses a_{k+1} rather than a, a_{k+1} and a has the same weight. We can form a new spanning tree T'' from T' by removing a and adding a_{k+1}. Then $T_{k+1} \subset T''$ which is also a MST.

Algorithm 1.7.4

Algorithm Kruskal's MST algorithm—Scheme:
$F = \emptyset$ # forest being constructed
$R =$ arcs(G) # remaining arcs
while R nonempty:
  remove a of smallest weight from R
  if a does not make a cycle when added to F:
    add a to F
return F

Proof

Let G be a connected weighted graph. We want to show that Kruskal?s algorithm constructs an MST for G.

Let G have n nodes. Each stage of the algorithm adds an arc to the subgraph constructed so far. Let the subgraphs constructed at each stage be $F_0,...,F_k,....$ We start with F_0 empty. Let F_{k+1} be got from F_k by adding arc a_{k+1}. We shall show by induction on k that each T_k is a subgraph of an MST T' of G.

Base case $k = 0$. T_0 is empty. Clearly $T_0 \subset T'$ for any MST T' of G.

If $a_{k+1} \in arcs(T')$ then $T_{k+1} \subset T'$ as required.

If $a_{k+1} \notin arcs(T')$. Since T' is a spanning tree, there must be a path P from x to y. There must be an arc a in P which does not belong to F_k, since otherwise the algorithm could not add a_{k+1} to F_k as it would form a cycle.

Since the algorithm chooses a_{k+1} rather than a, a_{k+1} and a has the same weight. We can form a new spanning tree T'' from T' by removing a and adding a_{k+1}. Then $T_{k+1} \subset T''$ which is also a MST.

Algorithm 1.7.5

Algorithm Dijkstra's Shortest Path Algorithm:
Input: Weighted graph (G,W) together with a pair of nodes start, finish
Output: Length of shortest path from start to finish
tree[start] = true
for x in adj[start]:
  # add x to fringe
  fringe[x] = true
  parent[x] = start
  distance[x] = W[start,x]
while finish not a tree node and fringe nonempty:
  Select a fringe node f such that distance[f] is minimum
  fringe[f] = false
  tree[f] = true
  for y in adj[f]:
    if not tree[y]:
      if fringe[y]:
        # update distance and candidate arc
        if distance[f] + W[f,y] < distance[y]:
          distance[y] = distance[f] + W[f,y]
          parent[y] = f
      else:
        # y is unseen
        fringe[y] = true
        distance[y] = distance[f] + W[f,y]
        parent[y] = f
return distance[finish]

Proof

We want to show if (G,W) is a connected weighted graph with nodes $start$ and $finish$, then Dijkstra's algorithm finds the shortest path from $start$ to $finish$.

We need to show that when a node f is added to the tree, distance[f] is the length of the shortest path.

Suppose we have a different and shorter path P (not necessarily in the tree). Let x be the last node on P to belong to the tree, and let y be the next node along P (possibly f itself). We know that shortest distance to $y \leq$ the distance from start to y using P. Hence the length of $P \geq$ shortest distance to y.

Since the algorithm chose f rather than y, then shortest distance to $y \geq$ shortest distance to f. Hence P is at least as long as path(f).

2 Algorithm Analysis

2.1 Searching an Unordered List

Algorithm 2.1.1

Algorithm Linear Search (LS):
$k = 0$
while $k < n$:
  if $L[k] == x$:
    return k
  else:
    $k = k + 1$
return "not found"

Proof

We want to show LS is optimal. Take any algorithm A which solves the problem. If A returns "not found" it must have inspected every entry of L. Suppose A did not inspect $L[k]$ (some $k < n$). We define a new list L' such that it is the same as L except that $L[k] = x$. Since A did not inspect $L[k]$ it will not inspect $L'[k]$ either, so A must still return "not found", which gives a contradiction. So A must inspect every element, meaning n comparisons are needed.

Definition 2.1.2

Worst case analysis: Let $W(n)$ be the largest number of comparisons made when performing LS on the whole range of inputs of size n. In this case $W(n) = n$.

Definition 2.1.3

Average case analysis: Let $A(n)$ be the average number of comparisons made when performing LS on the whole range of inputs of size n.

2.2 Searching an Ordered List

Algorithm 2.2.1

Algorithm Modified Linear Search (MLS):
$k = 0$
while $k < n$:
  cond:
    $L[k] = x$: return k
    $L[k] > x$: return "not found"
    $L[k] < x$: $k = k + 1$
return "not found"

$W(n)$ is still n as we may have to inspect the whole list.

Algorithm 2.2.2

Algorithm Binary Search (BS):
procedure BinSearch(left, right):
  # searches for x in L in the range L[left] to L[right]
  if left > right:
    return "not found"
  else:
    mid := $\lfloor$(left + right)/2$\rfloor$
  cond
    $x = L$[mid]: return mid
    $x < L$[mid]: BinSearch(left, mid − 1)
    $x > L$[mid]: BinSearch(mid + 1, right)

Since BS is recursive, we cannot find $W(n)$ directly. However we can write down the following recurrence relation $W(1) = 1$ and $W(n) = 1 + W(n/2)$. The number of 1s is the number of times that we can divide n by 2. Since $W(1) = 1$, $W(n) = 1 + \log_2 n$

Theorem 2.2.3

Any algorithm for searching an ordered list of length n for element x, and which only accesses the list by comparing x with entries in the list, must perform at least $\log n + 1$ comparisons in worst case.

Proof

Consider any algorithm A for searching an ordered list of length n. We can represent it by a binary decision tree. The tree will have at least n nodes since the algorithm must have the capability to inspect every member of the list. Let the depth of the tree be d, the worst-case performance of A will be $d + 1$. (d starts from 0).

Suppose that T is a binary tree with depth d. Then T has no more than $2^{d+1} - 1$ nodes. Base case $d = 0$.Then T has at most 1 node and $1 \leq 2^{0+1} - 1$. Assume true for all numbers $\leq d$. Let T be a tree with depth $d + 1$, then the two subtrees T_1 and T_2 both have depth $\leq d$, so they have less than $2^{d+1} - 1$ nodes. T has $1 + 2(2^{d+1} - 1) = 2^{(d+1)+1} - 1$ nodes. If T has n nodes and depth d then $n \leq 2^{d+1} - 1$. So $d + 1 \geq \log_2(n + 1)$.

$\log_2(n + 1) = \log_2 n + 1$. We see that the worst case behaviour of A (namely $d + 1$) is at least $log_2 n + 1$.

2.3 Orders of Complexity

Definition 2.3.1

f is **O(g)**("f is big Oh of g") iff there are $m \in \mathbb{N}, c \in \mathbb{R}^+$ such that $\forall n \geq m$ we have $f(n) \leq c \cdot g(n)$.

2.4 Sorting

Algorithm 2.4.1

Algorithm Insertion Sort:
$i = 1$
while $i < n$:
  # insert $L[i]$ into $L[0..i-1]$
  $j = i$
  while $L[j-1] > L[j]$ and $j > 0$:
    Swap$(j-1, j)$
    # swaps $L[j-1]$ and $L[j]$. $L[j]$ will always be the value to be inserted
    $j = j-1$
  $i = i+1$

The insertion can be done by comparing $L[i]$ successively with $L[i-1], L[i-2], \ldots$ until we find the first $L[j]$ such that $L[i] \geq L[j]$.Therefore in worst case we need to perform i comparisons for $i = 1, 2, \ldots, n-1$. $W(n) = n(n-1)/2$. So we have a sorting algorithm which is $O(n^2)$.

Algorithm 2.4.2

Algorithm Mergesort:
procedure Mergesort(left, right):
if left < right:
  mid $= \lfloor$(left + right)$/2 \rfloor$
  # left $\leq$ mid < right
  Mergesort(left, mid)
  Mergesort(mid + 1, right)
  Merge(left, mid, right)

To calculate $W(n)$, we use the following recurrence relation, $W(1) = 0$ and $W(n) = n - 1 + W(n/2) + W$ where $n-1$ is the number of comparisons to perform a merge. Assume $n = 2^k$.

$$
\begin{aligned}
W(n) &= n - 1 + 2W(n/2) \\
&= n - 1 + 2(n/2 - 1) + 2^2 W(n/2^2) \\
&= n + n - (1 + 2) + 2^2 W(n/2^2) \\
&= n + n + n + n \ldots - (1 + 2 + 2^2 + 2^3 + 2^{k-1}) + 2^k W(n/2^k) \\
&= k \cdot n - (n-1) + 0 \qquad\qquad 1 + 2 + 2^2 + 2^3 + 2^{k-1} = 2^k - 1 \\
&= n \log_2 n - 2^{\log_2 n} + 1
\end{aligned}
$$

Algorithm 2.4.3

Algorithm Quicksort:
procedure Quicksort(left, right):
if left < right:
  s = Split(left, right)
  Quicksort(left, $s-1$)
  Quicksort($s+1$, right)

Quicksort is harder to analyse than Mergesort because the list can be split into sublists of varying lengths. The worst case is when the split occurs at one end. This gives us the following recurrence relation. $W(1) = 0$ and $W(n) = n - 1 + W(n-1)$. Which is $W(n) = n(n-1)/2$.

Quicksort has a significant improvement for average-case complexity. When $Quicksort(0, n-1)$ is invoked it will split the list at some position s. We assume that s is equally likely to take each of the possible values $0, ..., n-1$. Each of these values therefore has a probability of $1/n$. If the split is at s, then $Quicksort(0, s-1)$ and $Quicksort(s+1, n-1)$ will be invoked, taking $A(s)$ and $A(n-s-1)$ comparisons respectively. Therefore $A(1) = 0$ and $A(n) = n - 1 + 1/n \sum_{s=0}^{n-1}(A(s) + A(n-s-1))$. Which can be simplified to

$$A(1) = 0$$
$$A(n) = n - 1 + \frac{2}{n}\sum_{i=2}^{n-1} A(i)$$

Theorem 2.4.4

Any algorithm for sorting a list of length n by comparisons must perform at least $\log(n!)$ comparisons in worst case.

Proof

Any algorithm for sorting by comparison can be expressed as a decision tree. Consider a decision tree for sorting a list of length n. It must have at least one leaf for every permutation of $[0, ..., n-1]$. Now there are $n!$ such permutations. So the tree needs at least $n!$ leaves.

If a binary tree has depth d then it has $\leq 2d$ leaves. Base case $d = 0$. Then no more than $1 = 2^0$ leaves. Assume true for all trees of depth $\leq d$. Suppose the tree has depth $d + 1$. The root has one or two successors, the successors have at most 2^d leaves each. So the total number of leaves in the tree $\leq 2^d + 2^d = 2^{d+1}$.

The worst-case performance of the algorithm is exactly the depth d of the tree. By the proposition, d must satisfy $n! \leq 2^d$. Hence $d \leq \log(n!)$.

3 Introduction to Complexity

3.1 Tractable Problems and P

Definition 3.1.1 A **decision problem** D is decided by an algorithm A if for any input x, A returns 'yes' or 'no' depending on whether $D(x)$ (and in particular A always terminates).

Definition 3.1.2

A decision problem D is **decidable** in polynomial time iff it is decided by some algorithm A which runs within polynomial time ($p-time$), i.e. on all inputs of size n, A takes $\leq p(n)$ steps for some $p(n)$.

tractable = polynomial time

Theorem 3.1.3

If a problem can be solved in $p-time$ in some reasonable model of computation, then it can be solved in $p-time$ in any other reasonable model of computation.

Definition 3.1.4

A decision problem $D(x)$ is in the **complexity class** P (polynomial time) if it can be decided within time $p(n)$ in some reasonable model of computation.

P class of decision problems which can be efficiently solved

Theorem 3.1.5

Suppose that f and g are functions which are $p-time$ computable. Then the composition $g \circ f$ is also $p-time$ computable.

Proof

Suppose that $f(x)$ is computed by algorithm A within time $p(n)$. $g(y)$ is computed by algorithm B within time $q(m)$. We compute $g(f(x))$ by first running A on x to get $f(x)$ and then running B on $f(x)$ to get $g(f(x))$. Running A on x takes $\leq p(n)$ steps, let's say it takes $p'(n)$ steps. Then B runs within $q(p'(n))$ steps. The total running time is $p(n)+q(p'(n))$. This is polynomial in n.

3.2 Complexity Class NP

Definition 3.2.1

A decision problem $D(x)$ is in NP **(Non-deterministic polynomial time)** if there is a problem $E(x,y)$ in P and a polynomial $p(n)$ such that $D(x)$ iff $\exists y.E(x,y)$ and if $E(x,y)$ then $|y| \leq p(|x|)$.

x is the original inputs for $D(x)$

y is the suggested proof for that $D(x)$ is true.

Note that we want y to be efficient in the size of x. If y is a big proof the verifier will be able to read only a polynomial part of y.

NP class of decision problems which can be efficiently verified.

Theorem 3.2.2

If a decision problem is in P then it is in NP, i.e. $P \subset NP$

3.3 Problem Reduction

Definition 3.3.1

Suppose that D and D' are two decision problems. We say that D **reduces** to $D(D \leq D')$ if there is a $p - time$ computable function f such that $D(x)$ iff $D'(f(x))$.

Suppose that we have an algorithm A' which decides D' in time $p'(n)$. Then if $D \leq D'$ via reduction function f running in time $p(n)$ we have an algorithm A to decide D:

Algorithm A (input x):
1. Compute $f(x)$
2. Run A' on input $f(x)$ and return the answer (yes/no)

Theorem 3.3.2

Suppose $D \leq D'$ and $D' \in P$. Then $D \in P$.

Theorem 3.3.3

Suppose $D \leq D'$ and $D' \in NP$. Then $D \in NP$.