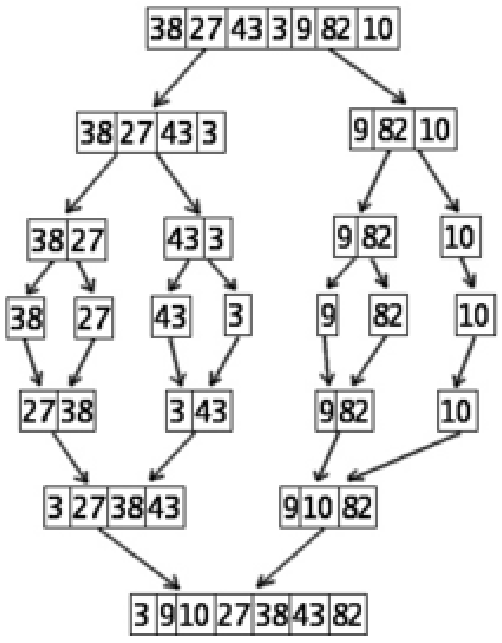
# Computer Science 577 Notes
# Introduction to Algorithms

Mendel C. Mayr

April 16, 2015



## Contents

# 1 Recurrence Relations

## 1.1 Recursive Analysis of Insertion and Merge Sort

Insertion sort: let $M(n)$ be the comparisons required to sort a list of size $n$
Analysis: note that $M(1) = 0$ and $M(n) = M(n-1) + n$ for $n > 1$

(i) $M(n) = M(n-1) + n$

(ii) $M(n) = M(n-2) + n + (n-1)$ ...

(iii) $M(n) = M(n-k) + n + (n-1) + \ ... \ + (n-k+1)$

(iv) Let $k = n-1$, $M(n) = M(1) + n(n-n+1) + \sum_{i=1}^{n-1} i$

(v) $M(n) = 0 + n + (n-1)(n-2)/2 \approx n^2/2$

Merge sort: let $M(n)$ be the comparison required to sort a list of size $n$

Analysis: for simplcity, consider only $n$ such that $n = 2^a$ for some integer $a$
Note that $M(1) = 0$ and $M(n) = 2M(n/2) + n$ for $n > 1$

(i) $M(n) = 2M(n/2) + n$

(ii) $M(n) = 2M(n/4) + n + (n/2)$

(iii) $M(n) = 2M(n/2^k) + n + (n/2) + \ .... \ + (n/2^{k-1})$

(iv) Let $k = a$, $2M(1) + \sum_{i=1}^{k-1} n/2^i$ (unclarified point)

(v) Mergesort is $O(n \log n)$

## 1.2 Recursive Linear Selection

Recursive linear selection algorithm: given $x_1, x_2, ..., x_n$ distinct keys, find $x_k$ (i.e. the $k$th smallest element) without using sorting
Note: the rank of an element (i.e. the number of keys greater than it) can be found in linear time
Linear selection algorithm is as folows:

(i) Remove keys of known rank, to make $n = 5(mod 10)$

(ii) Divide elements into groups of 5, denoted $S[i]$ for $i$ from 1 to $n/5$

(iii) Recursively find the median of each group, denoted $x[i]$

(iv) Let $M^*$ be the median of the set $x[i]$ for $i$ from 1 to $n/5$

(v) Divide keys into groups of keys less than (call this $L$), equal to, or greater than (call this $R$) $M^*$

(vi) Recursiveley process one of $L$ or $R$

Analysis: Note that steps 1, 2, and 5 are $O(n)$, so the number of computations for this algorithm, $T(n) = T(n/5) + T(7n/10) + O(n)$
Guessing and proving: supposed that $T(n) = O(n)$, which can be proven via strong induction, i.e. $T(n) < An$ for some constant $A$ for all $n$
Recall that strong induction relies on proving two claims:

(i) The statement holds for all $n \geq 1$, $\forall n \leq n_0$ (base case)

(ii) If the statmenet holds for all $i < n$, it holds for $n$

Proof of second part of strong inductive proof:

(i) Suppose that $T(n) = O(n)$ for $i < n$

(ii) We seek and $A$ such that $A(n/5) + A(7n/10) + cn \leq An$

(iii) Thus, $A \geq 10c$ is sufficient for this part

Proof of first part of strong inductive proof: need $A$ such that $n(n-1)/2 \leq An$ for $1 \leq n \leq 10$
So $A \geq 9/10$ is sufficient for this part

Conclusion: $A = max\{9/2, 10c\}$ will suffice to show that $T(n) = O(n)$

## 1.3 Recursive Quadratic Closest-Pair

Recursive quadratic algorithm: find closest pair of points

i (Supposing that $n = 2^k$) into 2 equal groups, denoted $L$ and $R$

ii Recurisvely find the closest pair in $L$ and $R$

iii Report closest pair form testing elements of $L$ against elements of $R$

iv Report best pair out of those from steps (ii) and (iii)

Analysis: $T(n) = 2(T/n) + O(n^2)$ for $n = 2^k \geq 4$, $T(2) = 1$
The $O(n^2)$ in the recursive case comes from step (iii)

Consider the recursion tree, which is full binary tree: at the first level, the problem size is $n, n/2, n/4, ...$ at the first, second, third, etc. levels. Thus, the number of computations required is $n^2$ at the first level, $2(n/2)^2 = n^2/2$ at the second level, $2(n/4)^2 = n^2/4$ at the third, etc.

Thus, the maximum number of computations is $\sum_{k=1}^{\infty} n^2/2^k = 2n^2$, thus $T(n) = O(n^2)$

## 1.4 Divide and Conquer Recurrences and Master Theorem

Master theorem: if $T(n) = aT(n/b) + O(n^d)$ for some constants $a > 0, b > 1$, and $d \geq 0$, then:

(i) $T(n) = O(n^d)$ if $d > \log_b a$

(ii) $T(n) = O(n^d \log n)$ if $d = \log_b a$

(iii) $T(n) = O(n^{\log_b a})$ if $d < \log_b a$

Proof: consider the recursion tree for such a problem
Notice that $a$ is the branching factor of the problem. At the $i$th level (starting at index 0), there are $a^i$ subproblems of size $n/b^i$. which means the computation that must be done at that level is $a^i O((n/b^i)^d)$
The number of levels in the recusion tree is $k = \log_b n$
As such: $T(n) = \sum_{i=0}^{k} a_i O((n/b^i)^d) = O(n^d) \sum_{i=0}^{k} (a/b^d)^i$. Now consider the cases

(i) If $d > \log_b a$, then $a/b^d < 1$
$\sum_{i=0}^{\infty} (a/b^d)^i < \infty$ (i.e. series converges)
$\sum_{i=0}^{\infty} (a/b^d)^i = O(1)$, so $T(n) = O(n^d)$

(ii) If $d = \log_b a$, then $a/b^d = 1$
$\sum_{i=0}^{k}(a/b^d)^i = \sum_{i=0}^{k} 1 = k+1$, since $k = \log_b n = \log n / \log b = O(\log n)$
Therefore, $T(n) = O(n^d \log n)$

(iii) If $d < \log_b a$, then $a/b^d > 1$
$\sum_{i=0}^{k}(a/b^d)^i = O((a/b^d)^k)$, so $T(n) = O(n^d)O(a^k)/b^{dk}$
Since $k = \log_b n, n = b^k$, $T(n) = O(n^d)O(a^k)/n^d = O(a^k)$
$a^k = a^{\log_b n} = n^{\log_b a}$, so $T(n) = O(n^{\log_b a})$

## 1.5 Asymptotics

Notation for asymptotics:

(i) $f$ and $g$ are real value functions on $x \geq 0$. $f(x)$, $g(x) \geq 0$ for sufficiently large $x$
Sufficently large: $\exists x_0 > 0$ such that $f(x) \geq 0$ when $x \geq x_0$

(ii) $f = O(g)$ means that for some $c > 0$ and $x_0 > 0$, $f(x) \leq cg(x)$ for al $x \geq x_0$

(iii) $f = \Omega(g)$ means that for some $c > 0$ and $x_0 \geq 0$, $f(x) \geq xg(x)$ for all $x \geq x_0$
This is equivalent to saying that $g = O(f)$

(iv) $f = \Theta(g)$ means that $f = O(g)$ and $f = \Theta(g)$

Demonstrating that $f = O(g)$ can be done algebraically, or via L'Hopital's rule

Additional definitions:

(i) $f = o(g)$ means $\lim_{x\to\infty} f(x)/g(x) = 0$

(ii) $f \sim g$ means $\lim_{x\to\infty} f(x)/g(x) = 1$

Polynomial growth: $f$ is polynomially bounded if $f(x) = O(x^k)$ for some $k > 0$, (efficiently computable)
Exponential growth: $f$ is exponential growth if $f(x) = O(\alpha^x)$ for some $\alpha > 1$

# 2 Arithmetic Algorithms and Sorted Lists

## 2.1 Arithmetic Algorithms

Addition: elementary (i.e. sum and carry bits), adding two $n$-bit numbers has $O(n)$ complexity
Subtraction: inverse of addition, similarly requires $O(n)$ times
Multiplication: elementary algorithm requires $O(n)$ times
There exists an $O(n^a)$ algorithm for multiplication, where $a < 2$:
For multiplying $n$-bit numbers, where $n = 2^k$:

(i) $x = 2^{n/2}x_1 + x_0$, $y = 2^{n/2}y_1 + y_0$

(ii) $xy = 2^n x_1 y_1 + 2^{n/2}(x_1 y_0 + x_0 y_1) + x_0 y_0$

(iii) Let $a = x_1 y_1$, $c = x_0 y_0$, $d = (x_1 + x_2)(y_1 + y_2)$

(iv) Let $b = x_1 y_0 + x_o y_1$, note that $b = D - A - C$

Analysis: let $k(n)$ be the complexity of this algorithm
$k(n) = 3k(n/2) + O(n)$ when $n > 1$, $K(n) = O(1)$ when $n = 1$
By the master theorem: $k(n) = O(n^{\log_2 3}) \approx O(n^{1.59})$

Using Newton interation, division reducible to multiplication: similar complexity
Open question: does there exist $O(n)$ algorithm for multiplication

Matrix multiplication: let $A = (a_{i,j}), B = (b_{i,j})$ for $1 \leq i, j \leq n$
Then $C = (c_{i,j})$, where $c_{i,j} = \sum_{k=1}^{n} a_{i,k} n_{k,j}$
Recursive algorithm for matrix multiplication: subblocks of matricies can be multipled as follows

$$\text{Since } X = \begin{bmatrix} A & B \\ C & D \end{bmatrix} \text{ and } Y = \begin{bmatrix} E & F \\ G & H \end{bmatrix}, \text{ then } XY = \begin{bmatrix} AE + BG & CE + DG \\ AF + BH & CF + DH \end{bmatrix}$$

The 8 products required tocan be found recursively, resulting in $T(n) = 8T(n/2) + O(n^2) = O(n^3)$
A matrix decomposition can be used to make a faster algorithm, requiring only 7 products:

$$\begin{bmatrix} P_5 + P_4 - P_2 + P_6 & P_1 + P_2 \\ P_3 + P_4 & P_1 + P_5 - P_3 - P_7 \end{bmatrix}$$

The products $P_1, P_2, ..., P_7$ are defined respectively as:
$A(F - H), (A + B)H, (C + D)E, D(G - E), (A + D)(E + H), (B - D)(G + H), (A - C)(E + F)$
The complexity, by the master theorem, is: $T(n) = 7T(n/2) + O(n^2) = O(n^{\log_2 7}) \approx O(n^{2.81})$

## 2.2 Quicksort

Quicksort algorithm: given array $E$

(i) Selects pivot element, moves element to local variable

(ii) Partition subroutine rearranges elements about a *splitPoint* such that:

  (a) For $first \leq i < splitPoint$, $E[i] < pivot$
  (b) For $splitPoint < i \leq last$, $E[i] \geq pivot$

(iii) Pivot element goes in $E[splitPoint]$

(iv) Recursively sort the smaler and larger subarrays

Analysis of quicksort:
Worst case: already sorted in ascending order, smallest element selected as pivot
Complexity in worst case is: $\sum_{k=2}^{n}(k-1) = n(n-1)/2$

Average behavior: suppose all permutations of keys are equally likely

(i) For an array of size $k$, partition does $k-1$ key comparisons
Subranges have $i$ and $k-1-i$ elements each

(ii) This gives the following recurrence: $A(n) = 0$ for $n = 1$ or $n = 2$
$A(n) = n - 1 + \sum_{i=0}^{n-1}(1/n)(A_i) + A(n-1-i)$ for $n \geq 2$
Which is that same as: $A(n) = n - 1 + (2/n)\sum_{i=1}^{n-1} A(i)$ for $n \geq 1$

(iii) A good case for quicksort is if each partition divides the array into 2 arrays of size $n/2$ each
In this case: $Q(n) \approx n + 2Q(n/2)$, so by the master theorem $Q(n) = \Theta(n \log n)$

Theorem: let $A(n)$ be defined by the recurrence as above. Then for $n \geq 1$, $A(n) \leq cn\ln(n)$ for some constant $c$

Proof: strong induction supposes that $A(i) \leq ci\ln(i)$ for $1 \leq i < n$
Thus, suppose that $A(n) \leq n - 1 + (2/n)\sum_{i=1}^{n-1} ci\ln(i)$
$A(n) \leq n - 1 + (2/n)\int_{1}^{n} x\ln(x)dx = cn\ln(n) + n(1 - c/2) - 1$
Let $c = 2$ so that $A(n) \leq 2n\ln(n)$. A similar analysis shows that $A(n) > cn\ln(n)$ for $c < 2$

Corrolary: average case of number of comparisons done by quicksort is $1.386n\log(n)$ for large $n$

## 2.3 Random Choices in Algorithms

Example: seearch aray for a 1, where $\Sigma = \{0, 1\}$, and array has 50% 0s
A deterministic algorithm will use a linear search, with worst case taking $n/2$ time
Let $T$ be the queries to find: $E(T) = \sum_{t=0}^{\infty} t/2^t = 2$

Example: Quicksort (with randomly chosen pivot) to sort distinct set $x_1, ..., x_n$
Choose $1 \leq i \leq n$ at random, let pivot $p = x_i$
Acts just like quicksort on a randomly ordered input

Skip lists: storing a list of distinct sorted numbers
Multilevel indicies, i.e. various elements stored in nodes, with 2-way connections between:

(i) Adjacent elements (connections between elements of the same level)

(ii) Identical elements (connections between various levels)

Tab locations (e.g. nodes at higher levels) made using random choices
Setinels: use before and after at each level (e.g. $\infty$ and $-\infty$)
To find value: start at highest level, decend right before element value is passed

The number of levels for any particular index is randomly determined
Number of nodes in skip list: $E(\#nodes) = \sum_{x \in keys} E(height)$, where $E(height) = 2$
The height of the skip list is based on a geometric random variable, i.e. height is flip number for first head
Analysis of skip lists: expected cost of search, insert, and delete is $O(\log n)$

## 2.4 Splay Trees

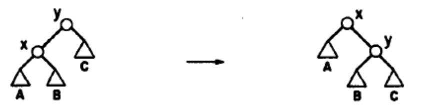
Amortized analysis: expected complexity value for a series of operations
Splaying: used to create logarithmic amortized bound

Splaying restructures tree to move latest node to the root

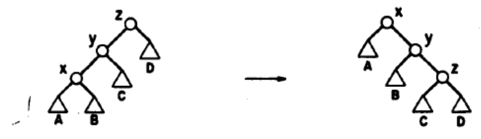
(a) Zig case: single rotation to put $x$ at root

(b) Zig-zag case: two single rotations, parent and grandparent each rotated

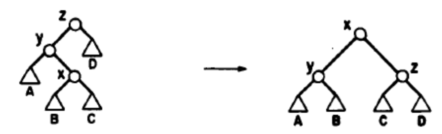
(c) Zig-zig case: double rotation, parent and grandparent rotated at once via $x$



(a)



(b)



(c)

Triangles in figure represent subtrees. Trees shown here can themselves be subtrees

Operations on splay trees: element $x$

(i) Insertion: splay upon insertion, $x$ becomes new root

(ii) Find: if search is successful, splay $x$ to new root, otherwise, last node access prior to reaching null value becomes new root

(iii) FindMin and FindMax: splay after access, $x$ moved to root

(iv) DeleteMin and DeleteMax:

  (a) DeleteMin: perform FindMin, brining minimum to root
  By binary search tree property, no left child. Use right child as new root

  (b) DeleteMax: perform FindMax, brining maximum to root
  By binary search tree property, no right child. Use left child as new root

(v) Remove: bring $x$ to the root and delete, leaving left and right subtrees $L$ and $R$
Use deleteMax to find largest element in $L$, leaving root of $L$ with no right child
Make $R$ the right child of $L$'s root

Key insight: any node at depth $d$ on access path gets moved to a new depth $d' \leq d/2 + 3 = d/2 + O(1)$
Top-down splay tree: at any point in middle of splay

(i) The current node $x$ is the root of the subtree

(ii) Tree $L$ stores nodes less than $X$

(iii) Tree $R$ stores nodes greater than $X$

Initially, $X$ is the root, and $L$ and $R$ are empty
Descend tree two levels at a time, encountering a pair of nodes:

(i) Nodes are placed in $L$ or $R$ depending on if they are smaller or larger than $x$

(ii) Subtrees not on access path to $x$ also put in $L$ and $R$ trees

(iii) When $x$ reached, attach $L$ and $R$ to bottom of middle tree

Potential function analysis: $\Phi$ maps state (e.g. of splay tree) to non-negative potential value
Potential function for splay tree: low for balanced trees, high for unbalanced trees

(i) $size(r)$ = number of nodes in subtree rooted at $r$ (including $r$)

(ii) $rank(r) = \log_2(size(r))$

(iii) $\Phi = \sum_r rank(r)$, i.e. the sum of rank for all nodes in the tree

# 3 Data Structures and Algorithms

## 3.1 Graphs and Graph Traversal

Depth-first search: relies on two abilites

(i) Marking visited nodes: maintain boolean for each vertex

(ii) Backtracking: use recursion stack for vertex backtracking

Depth-first search repeatedly explores unvisited nodes until entire graph traversed
The exploration of each vertex involves the following steps:

(i) Fixed amount of work to mark vertex, and pre/postvisit, $O(|V|)$ time total

(ii) Iterating over adjacent edges to check for unvisited vertices, $O(2|E|)$ time total

Thus, to traverse the graph $(V, E)$ requires time $O(|V| + |E|)$, i.e. linear in size of the input
An undirected graph is connected if there is a path between any pair of veritices
Connected components: connected subsets of vertices of a graph

Breadth-first search: implemented using a queue, initially containing the start node, operates in $O(|V|+|E|)$ times, operating similarly to depth-first search
Minimum spanning trees (MST):
Minimum spanning tree of graph $G = (V, E)$ with edge weights $w_e$: $T = (V, E' \subseteq E)$ minimizing $\sum_{e \in E'} w_e$
Kruskal's minimum spanning tree algorithm: repeatedly add next lightest edge that doesn't produce a cycle

Correctness of Kruskal's algorithm follows from cut property: suppose edges $X$ are part of a MST of $G = (V, E)$. Then pick any $S \subset V$ for which $X$ does not cross betweeen $S$ and $V - S$ and let $e$ be the lightest edge across the partition. Then $X \cup \{e\}$ is part of some MST
Cut: any partition of vertices into group $S$ and $V - S$. The cut property states that the lightest edge across any cut is in the MST, provided $X$ have no edges across the cut

Prim's algorithm:

(i) Initialize $V_{new} = \{x\}$, where $x$ is an arbitrary starting point, $E_{new} = \{\}$

(ii) While $V_{new} \neq V$: find minimal edge $(u, v)$, where $u \in V_{new}$ and $v \in V - V_{new}$, then add $v$ to $V_{new}$ and $(u, v)$ to $E_{new}$

(iii) Output minimum spanning tree: $T = (V_{new}, E_{new})$

## 3.2 Shortest Path Problems

Dijkstra's algorithm: breadth-first search using priority queue, given $G = (V, E)$ and lengths $w_e$
Algorithm uses distance measure $d(v)$, the shortest distance from start to $v$, and $p(v)$, the parent of $v$

(i) For all $v \in V$, $d(v) = \infty$ and $p(v) = null$, and let $d(s) = 0$

(ii) Create a priority queue from $V$, using $d(v)$ as keys

(iii) While priority queue is not empty

   (a) Let $u$ be from popping the min element of priority queue

   (b) For all edges $(u, v) \in E$: if $d(v) > d(u) + w_{(u,v)}$, then set $d(v) = d(u) + w_{(u,v)}$
   Set $p(v) = u$, then update the priority queue

Priority queue implementations:

(i) Array: unordered array of key values for elements, with initial values set to $\infty$

(ii) Binary heap: complete binary tree, all operations $\log |V|$

(iii) $d$-ary heap: heap where nodes have $d$ children

## 3.3 Cycle Detection Problem

Depth-first search in directed graphs: consider the types of edges

(i) Tree edges: part of the DFS forest, connect parents and children

(ii) Back edges: lead a vertex to its ancestor in DFS tree

(iii) Forward edges: lead from vertex to a nonchild descendant in DFS tree

(iv) Cross edges: lead to neither descendants nor ancestors (i.e. postvisted vertices, new to old)

A directed graph has a cycle if and only if its depth-first search reveals a back egde
To dinstinguish between edges, use timing information: for each node visitation

(i) Set $pre[v] = clock++$ before checking children, where $clock$ is global clock

(ii) set $post[v] = clock++$ after checking children (i.e. exploring subtree)

$[pre[v], post[v]]$ is the interval in which the exploration subroutine for $v$ is active
Note that intervals for a descendant are nested in intervals for their ancestors
Consider the types of edges and the possible nestings for edge $(u, v)$

(i) Tree/forward: $pre[u] < pre[v] < post[v] < post[u]$, i.e. $[_u [_v ]_v ]_u$

(ii) Back: $pre[v] < pre[u] < post[u] < post[v]$, i.e. $[_v [_u ]_u ]_v$

(iii) Cross: $pre[v] < post[v] < pre[u] < post[u]$, i.e. $[_v ]_v [_u ]_u$

## 3.4 Sources and Sinks in Directed Acyclic Graphs

Sources have no inbound edges, sinks have no outbound edges
Note that every nonempty directed acyclic graph has at least 1 source and 1 sink
All directed acyclic graphs can be linearlized: if there's a path from $v$ to $w$, $v$ is before $w$
Linearization algorithm: find a source $s$, then recursively linearlize $G = (V - s, E)$
Fast linearlization algorithm: obeserve that $post[v]$ decreases along each edge
Use array to sort post numbers in range 1 to $2|V|$. Put vertex with $post[v] = n$ into $n$th cell

Strongly-connected components (SCC): strongly connected, disjoint sets of vertices, each a subset of $V$
If each SCC is collapsed to a single vertex (removing redundant edges) a DAG is the result
Finding a source: largest post number from DFS is in vertex of source component
Reverse the edges of the graph to find a sink component by this method
Kosaraju/Sharir Algorithm: $O(|V| + |E|)$

(i) Run full depth-first search on $G^R$, assign post numbers to all vertices

(ii) While there exist unassigned $v \in V$

    (a) Set $v$ to the assigned vertex with the largest post number
    (b) Run depth-first search from $v$ in $G$, assigning all reachable vertices to $v$'s component

(iii) Remove $v$ from the graph

## 3.5 Self Organizing Lists

Average case: $E(t) = \sum_{i=1}^{n} ip(i)$, where $p(i)$ is access probability
Self-organizing lists are rearranged so more likely accessed elements have lower access time

Techniques for rearranging nodes:

(i) Move to front method (MTF): any node accessed is moved to the front of the list

(ii) Count method: each node stores number of times accessed, nodes ordered by decreasing count

(iii) Tranpose method: any node accessed is swapped with preceding node

## 3.6 Union-Find Data Structure

Union-Find data structure: maintains a partition of the universe into disjoint sets
Structure contains elements, partitioned into disjoint subsets, each with representative element
To test if elements are in the same subset, compare their representative values

Union-Find data structure has 3 operations:

(i) Makeset: creates singleton set with specified element

(ii) Find: determine the subset of an element, returns a representative element

(iii) Union: join two subsets into a single subset
Implement by taking root of one element and point it to root of the other

Can be implemented using trees, with each tree being a subset and the root representing

Rank: the rank of $x$ is the height of $x$'s subtree
Union by rank: choose the root of larger rank during union operation
Note that:

(i) Rank increases along a path to root

(ii) Any rank-$k$ node has $\geq 2^k$ nodes in its subtree

(iii) If we have $n$ elements total, $n_k = \#$ of rank $k$ nodes $\leq n/2^k$

Thus, the complexity of operations are: $O(1)$ for $MakeSet$, $O(\log n)$ for find and union

# 4 NP-Complete Problems

## 4.1 NP-Complete Graph Problems

Class P: Decidable in polynomial time, i.e. $O(n^k)$ for some $k \geq 1$
Class NP: Certificate decidable in polynomial time, i.e. solution recognizable in polynomial time
NP complete: two requirements for an NP complete problem

(i) NP-complete problems are in NP

(ii) All NP-complete problems reducible to an NP-complete problem

Cook-Levin theorem: SAT (Boolean satisfiability) is NP-complete
By extension, 3CNFSAT is NP-complete
Three graph-theoretic NP-complete problems: given graph $G$, number $n$

(i) Vertex cover: set of vertices $n$ that touch all edges

(ii) Independent set: set of vertices $n$, none of which joined by an edge

(iii) Clique: set of $n$ fully connected vertices

3CNFSAT is reducible to vertex cover:

(i) For each boolean variable $x$, create vertices $x$ and $\neg x$, joined by edge

(ii) For each clause, three connected vertices, representing each literal

(iii) Literals connected to corresponding clause gadgets

(iv) For formula with $n$ variables and $c$ clauses, graph has $2n + 3c$ vertices

(v) Formula is satisfiable if and only if there is a vertex cover of size $n + 2c$
$n$ vertices used to cover literal gadgets, $2c$ vertices used to cover clause gadgets

Independent set reducible to vertex cover: for graph $G = (V, E)$, there is a vertex cover of size $n$ if and only if for the complement of $G$ (opposite edge presence), there is an independent set of size $|V| - n$
For graph $G = (V, E)$, $S$ is an independent set if and only if $V - S$ is a vertex cover, since for each edge $(u, v)$, only one of $u, v$ is in $S$, which means at least one of $u, v$ is in $V - S$

Clique to vertex cover: for graph $G = (V, E)$ and complement $G' = (V, E')$, $G$ has clique of size $k$ if and only if $G'$ has vertex cover of size $|V| - k$

# 5 Dynamic Programming

## 6 Network Flow and Linear Programming

# A Review of Basic Mathematical Concepts

## A.1 Properties of Logarithms

Change of base: $\log_a x = \log_b x / \log_b a$
Basic properties:

(i) $\log_a(uv) = \log_a u + \log_a v$

(ii) $\log_a(u/v) = \log_a u - \log_a v$

(iii) $\log_a u^n = n \log_a u$