CS 341

These notes are meant to be supplementary to lecture slides & the textbook, and so may not contain all covered materials. Here I've chosen content which might not be easy to remember and/or is helpful to look at when doing assignments.

Asymptotic Analysis

- **Order Notations**:
    - $f(n) \in O(g(n))$ if $\exists\ c > 0$ and $n_0 > 0$ such that $0 \leq f(n) \leq cg(n)\ \forall\ n \geq n_0$
        - f "grows no faster than" g
    - $f(n) \in \Omega(g(n))$ if $\exists\ c > 0$ and $n_0 > 0$ such that $0 \leq cg(n) \leq f(n)\ \forall\ n \geq n_0$
        - f "grows no slower than" g
    - $f(n) \in \Theta(g(n))$ if $\exists\ c_1, c_2 > 0$ and $n_0 > 0$ such that $0 \leq c_1 g(n) \leq f(n) \leq c_2 g(n)\ \forall\ n \geq n_0$
        - f ang g have the same complexity
    - $f(n) \in o(g(n))$ if $\forall\ c > 0, \exists\ n_0 > 0$ such that $0 \leq f(n) < cg(n)\ \forall\ n \geq n_0$
        - f has lower complexity than g
    - $f(n) \in \omega(g(n))$ if $\forall\ c > 0, \exists\ n_0 > 0$ such that $0 \leq cg(n) < f(n)\ \forall\ n \geq n_0$
        - f has higher complexity than g
    - $f \in O(g)$ and $f \in \Omega(g) \iff f \in \Theta(g)$
- **Limit method**: suppose $L = \lim_{n\to\infty} \dfrac{f(n)}{g(n)}$
    - $f \in o(g)$ if $L = 0$
    - $f \in \Theta(g)$ if $0 < L < \infty$
    - $f \in \omega(g)$ if $L = \infty$
- Useful facts for first-principles proofs:
    - $\log n \geq 1\ \forall\ n \geq 2$; i.e. $\log n$ grows faster than 1
    - $\log n \leq n\ \forall\ n \geq 0$; i.e. $\log n$ grows slower than n
- Useful limit laws:
    - $\lim_{n\to\infty} \dfrac{f(n)}{g(n)} = \lim_{n\to\infty} \dfrac{\log(f(n))}{\log(g(n))}$
    - $\lim_{n\to\infty} f(n)^k = (\lim_{n\to\infty} f(n))^k$
- Some math rules:
    - Summing a polynomial: $\sum_{i=1}^{n} i^k \in \Theta(n^{k+1})$

- Summing an exponential (special case of geometric series):

$$\sum_{i=1}^{n} c^i \in \begin{cases} \Theta(c^{n+1}) & \text{if } c > 1 \\ \Theta(n) & \text{if } c = 1 \\ \Theta(1) & \text{if } c < 1 \end{cases}$$

- $a^{\log_b n} = n^{\log_b a}$ (Useful for recursion trees)
- Geometric series:

$$\sum_{i=0}^{n-1} ar^i = \begin{cases} a\frac{r^n-1}{r-1} \in \Theta(r^n) & \text{if } r > 1 \\ na \in \Theta(n) & \text{if } r = 1 \\ a\frac{1-r^n}{1-r} \in \Theta(1) & \text{if } r < 1 \end{cases}$$

- $a \geq b + c$ if $a \geq 2 \cdot \max(b, c)$ (useful for asymtotic proofs)
- $\lim_{x \to c} \dfrac{f(x)}{g(x)} = \lim_{x \to c} \dfrac{f'(x)}{g'(x)}$ if it exists (L'Hopital's Rule)
- $\log(n!) \in \Theta(n \log n)$ (Stirling's Approximation)
- $\sum_{i=1}^{n} \dfrac{1}{i} \in \Theta(\log n)$ (Harmonic series)

Divide and Conquer

- **Recursion-tree method**:
  - Given the recurrence $T(n) = aT(n/b) + f(n), T(1) = c$:
    - a is the # of recursive calls made (# of subproblems)
    - b is the # by which the input size n is divided in each recursive call
    - $f(n)$ is the runtime of the "work done outside of the recursive calls"
    - c is the constant-time work done in each recursive call in the base case
  - Each node of the recursion tree represents the cost of the work done other than making recursive calls
  - Each row represents the total cost of work done in all recursive calls at that recursion "level"
  - The height of the tree depends on the factor that the input size is divided by; i.e. $\log_b n$
  - E.g.: Picture this as a tree where each node (except for leaves) has a children;

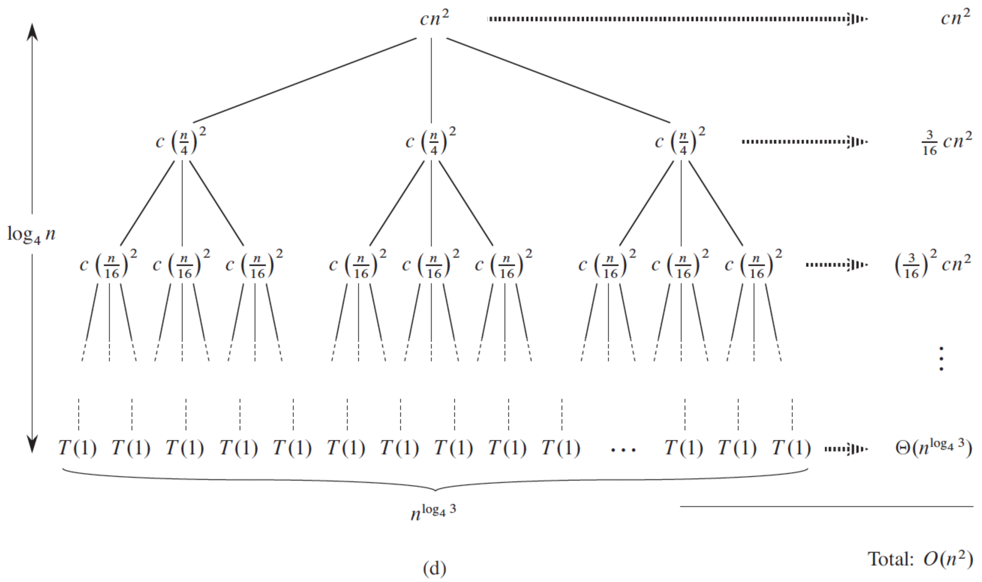
Level 0:	$f(n)$			total $= f(n)$
Level 1:	$f(n/b)$	$\ldots$	$f(n/b)$	total $= af(n/b)$
Level 2:	$f(n/b^2)$	$\ldots$	$f(n/b^2)$	total $= a^2 f(n/b^2)$
$\ldots$				
Level k:	$f(n/b^k)$	$\ldots$	$f(n/b^k)$	total $= a^k f(n/b^k)$
$\ldots$				
Level $\log_b n$:	c	$\ldots$	c	total $= ca^{\log_b n} = cn^{\log_b a}$

    - Total runtime of recursion tree is (summing every row total):

$$T(n) \in \Theta\left(\sum_{i=0}^{\log_b n - 1} a^i f(n/b^i)\right) + \Theta(n^{\log_b a})$$

    - Use geometric series formula to find a simplified value

- Example: $T(n) = 3T(n/4) + n^2$



(d)

$$T(n) \in \Theta\left(\sum_{k=0}^{\log_4 n-1} \left(\frac{3}{16}\right)^k n^2 + n^{\log_4 3}\right) \in \Theta(n^2)$$

- **Master method**:
  - Given the recurrence $T(n) = aT(n/b) + f(n)$ where $f(n) \in \Theta(n^d)$:

$$T(n) \in \begin{cases} \Theta(n^{\log_b a}) & \text{if } a > b^d \\ \Theta(n^d \log n) & \text{if } a = b^d \\ \Theta(n^d) & \text{if } a < b^d \end{cases}$$

Greedy Algorithms

- Two methods of proving greedy algorithms:
- **Greedy stays ahead (induction)**
  - Show that S_g is better than S at every step
  - Base case: show $S_g[1] > S[1]$
  - Inductive hypothesis: assume $S_g[k-1] > S[k-1]$; show that $S_g[k] > S[k]$
    - Often by contradiction; i.e. suppose $\exists\, S^*$ such that $S_g[k] < S^*[k]$, and derive some contradiction
- **Exchange argument (swapping)**
  - Let S_g = greedy solution, S = some arbitrary solution
  - Show that (any) S can be transformed into S_g step-by-step without getting worse at any point
    - i.e. compare the cost of before & after swapping 2 elements in S, show that it doesn't change or improves

Graphs

- **Breadth-first search**

```
function BFS(G, s) {        // s = starting vertex
    Q = queue
    mark all vertices as unvisited
    mark s as visiting
    Q.enqueue(s)
    while (Q not empty):
        v = Q.dequeue()
        for each unvisited neighbour u of v:
            mark u as visiting
            Q.enqueue(u)
        mark v as finished
}
```

    - Forward edges cannot exist in BFS trees; cross & back edges can
    - Cross edges indicate presence of an odd cycle – graph is not bipartite

- **Depth-first search**

```
function DFS-visit(G, s) {  // s = starting vertex
    mark s as visiting
    for each unvisited neighbour u of s:
        DFS(G, u)
    set s as finished
}

function DFS(G) {           // creates DFS forest
    mark all vertices as unvisited
    for each unvisited v in G.V:
        DFS-visit(G, v)
}
```

    - Iterative implementation using a stack is also possible

- **Topological sorting**
    - Order the nodes such that for every $(u, v) \in E$, u precedes v
    - Algorithm 1: remove source nodes iteratively
    - Algorithm 2: run DFS with finishing times of each node, then order by decreasing finish time
- **Strongly connected components**
    - An SCC = a *maximal* subset of vertices such that every pair u, v are reachable from each other
        - i.e. $\exists$ path from $u \to v$ and $v \to u$

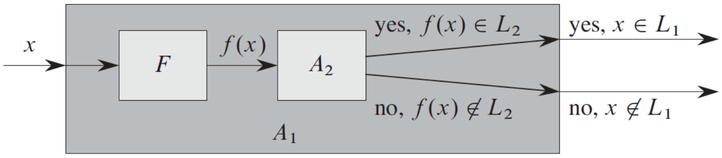
  - The component graph of G, G^{SCC}, is a DAG
  - **Kosaraju's Algorithm**:

```
function find_SCCs(G) {
    compute G_rev
    DFS(G) -> keep track of finish times
    for each unvisited v in G.V, by decreasing finishing time:
        DFS(G, v) -> label as SCC
}
```

- **Minimum spanning tree**:
  - Sum of edge weights is minimal
  - General greedy MST algorithm:
    - Add the minimum weighted edge crossing the cut formed by the tree "so far" & the rest of the graph
    - Repeat for $n - 1$ times
  - **Prim's Algorithm**: general greedy algorithm but with heap-based priority queue
    - Runtime = $O(m \log n)$
  - **Kruskal's Algorithm**:
    - Sort edges by weight; choose the minimum weighted edge that doesn't create a cycle
    - Runtime = $O(m \log n)$
- **Shortest path**:
  - SSSP in DAG:
    - Use DP, and the recurrence: $SD[v] = min_{(u,v) \in E} SD[u] + w(u, v)$
    - i.e. subproblem is the SSSP to the vertices preceding v
  - **Dijkstra's Algorithm**:
    - At each step, pick the vertex with the shortest distance so far
    - "Overwrite" distance-so-far of a vertex if a shorter path to it has been found
    - Runtime = $O(m \log n)$
  - **Floyd-Warshall's Algorithm** (all-pairs shortest path):
    - $D[i, j, k]$ = shortest path from i to j that only goes through $v_1 \dots v_k$ as intermediate vertices
    - Recurrence: $D[i, j, k] = \min(D[i, j, k - 1], D[i, k, k - 1] + D[k, j, k - 1])$
    - Runtime = $O(n^3)$

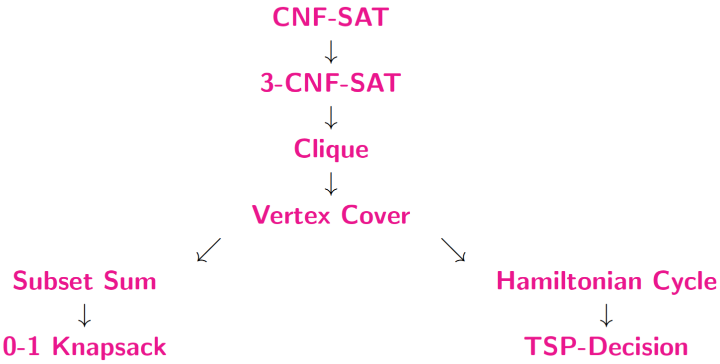
Intractability

- P: solvable in polynomial time
- NP: verifiable in polynomial time
    - A verification algorithm takes 2 inputs:
    - Problem instance – a particular set of parameters of the problem
        - e.g. a graph, with vertices and edges
    - Certificate – a given “solution” of the problem that shows the instance returns *yes*
        - e.g. a ordered list of vertices, used to verify that a Hamiltonian cycle indeed exists
- $P \subseteq NP$
- **Reducibility**:
    - $L_1 \leq_P L_2$ – “L_1 is polynomial-time reducible to L_2”
    - There exists a P-time computable function f that maps instances of L_1 to L_2
    - e.g. If problem instance I returns *yes* for L_1, then $f(I)$ also returns *yes* for L_2



    - If $L_1 \leq_P L_2$, then $L_2 \in P \implies L_1 \in P$
        - i.e. if L_2 is solvable in P-time, then L_1 is as well
        - i.e. L_1 is no harder than L_2
    - Contrapositive:
        - i.e. if L_1 is known to be *not* solvable in P-time, then L_2 can’t be either
        - i.e. L_2 is as hard as L_1
    - Mainly used for reducing decision problems as part of NPC proofs
- **Turing reduction**:
    - A reduction $L_1 \leq^T L_2$ used to solve L_1, where L_2 is known, e.g. available as a subroutine
    - $L_1 \leq_P^T L_2$ – polynomial-time Turing reduction
    - L_1, L_2 don’t have to be decision problems
    - Mainly used for reducing optimization problem → decision problem
- **NP-Completeness**:
    - L is NPC if:
        - $L \in NP$, and
        - $\forall\ L' \in NP, L' \leq_P L$ (this is definition of NP-hard)
    - i.e. NPC = set of problems in NP such that all other NP problems can be can be

    reduced to X
  - NPC = NP $\cap$ NP-hard
  - Because of these properties, to prove a problem L is NPC:
    - Show that $L \in \text{NP}$
    - Show that $L' \leq_P L$ for some $L' \in \text{NPC}$
- Reduction relationships between NPC problems:



- **Undecidability**:
  - No algorithm can solve the problem
  - Show that the existence of an algorithm leads to a contradiction
  - Example: the Halting Problem
  - Reduce an undecidable problem to another problem to show that it's also undecidable